Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Novel Quality Assurance Approaches for Web and IoT Systems

by

Diego Clerissi

Theses Series **DIBRIS-TH-2020-XX**

DIBRIS, Università di Genova
Via Opera Pia, 13 16145 Genova, Italy http://www.dibris.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

Novel Quality Assurance Approaches for Web and IoT Systems

by

Diego Clerissi

May, 2020

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
http://www.dibris.unige.it/

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum
(S.S.D. INF/01)

Submitted by Diego Clerissi
DIBRIS, Univ. di Genova
diego.clerissi@dibris.unige.it

Date of submission: March 2020

Title: Novel Quality Assurance Approaches for Web and IoT Systems

Advisors: Prof. Filippo Ricca, Dr. Maurizio Leotta
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova
filippo.ricca@unige.it, maurizio.leotta@unige.it

Ext. Reviewers: Prof. Damiano Distante, Dr. Angelo Susi

Abstract

For years, web-based systems have supported our daily activities by keeping us always connected to the world. With the emerging Internet of Things (IoT) technology, the improvements to our lives have been pushed further, enabling physically and virtually interconnected devices to share data and offer a myriad of disparate services. To address the continuously growing customers' expectations and the natural software evolution, both web and IoT based systems must undergo through rapid development cycles, supported by effective quality assurance strategies.

In the web domain, agile approaches are considered appropriate to respond to fast requirements changes, in particular, the Acceptance Test Driven Development (ATDD) practice, which puts the testing activity on top of the software development process. However, applying ATDD can be difficult in practice, since existing web testing tools require an underlying running web application, which is usually missing at such early software development stage, without mentioning the non-trivial manual skills required to convert abstract test cases into executable test scripts.

In the IoT domain, Node-RED recently emerged as a visual-based tool to offer a simple interface as a response to the complexity of developing and deploying IoT systems, that often employ a plethora of different devices, each one to be configured separately. However, despite Node-RED claimed simplicity, assuring the quality of IoT systems is still an open problem and no consolidated approaches involving Node-RED exist.

During my PhD, I have considered and investigated some of the aspects pertaining the quality assurance of web and IoT based systems, in particular when a requirements specification is introduced as a backbone to drive the whole process.

Concerning the web domain, an ATDD approach has been proposed for developing and testing web applications, where the requirements specification, in the form of precise use cases, are enriched with HTML screen mockups representing the web application prototype, exposing from a functional point of view all its main

functionalities. The developers and testers can then replicate the actions described in the use cases over the screen mockups, by recording them with any capture-replay web testing tool, to produce a raw set of executable test scripts driving the development and refinement stages. The approach has been applied to re-develop the main features of an open-source web application. To overcome the limitations that emerged during its application, such as the manual activity required to generate and maintain the test scripts, during the natural requirements evolution, a second approach has been sketched. This approach enforces the structure of the requirements specification with a UML model, and proposes a set of transformations to exploit the existing tools to semi-automatically generate the test scripts from the UML model and keep all the artifacts synchronized.

Concerning the IoT domain, a set of guidelines has been proposed to support the development of IoT systems in Node-RED and solve several well-known comprehensibility issues of the environment, taking inspiration from general design principles. The guidelines have been empirically evaluated by means of an experiment involving a class of Computer Science master students, showing that their application effectively reduces the number of errors and the time required to comprehend Node-RED systems. Moreover, an approach has been conceived to semi-automatically generate Node-RED artifacts and test scripts from the UML model of the static and dynamic properties of an IoT system. Finally, an acceptance testing approach for IoT systems has been proposed, where a UML state machine is used to describe the expected behavior of the system and drives the development in Node-RED and the testing phases; the acceptance testing approach has been applied on a realistic case study, a mobile health IoT system for diabetes management composed of heterogeneous devices, and compared against an existing runtime verification approach, in terms of strengths and weaknesses. The approach was able to detect between 71% to 100% of the types of bugs injected in the system.

Table of Contents

Part I

Introduction

Chapter 1

Thesis Background, Challenges, and Achievements

This thesis presents the main achievements of my PhD career at the University of Genova.

Before my first PhD year, I was involved by the Software Engineering Research Group of the Department of Computer Science at the University of Genova in several research activities pertaining web testing (e.g., [LCRT13, LCRS13b, LCRT14]) and modeling methods and notations (e.g., [RLRC15, RLRC14]). Web testing, in particular, focused on regression testing, empirical studies, and comparisons of the state of the art test frameworks. Instead, the modeling methods and notations research activities were oriented to investigate the knowledge and usages by professionals and users of the Unified Modeling Language (UML) artifacts and tools.

The importance of web testing became more and more evident during my starting research work, due to the ubiquitous involvement of the web in any kind of modern activity and its very rapid technological evolution, requiring novel testing strategies to be always kept in step with the target web applications. Given this background, and supported by my future advisors, I identified as a possible research topic for my forthcoming PhD career the formulation of novel techniques oriented to quality assurance of web applications, taking in consideration the problem of fast changes in requirements, by using a requirements specification as a backbone to drive test artifacts generation. The original intention was to propose (and refine) an approach to generate effective test cases strongly linked to the requirements and naturally oriented to evolve, when the associated requirements change.

As my first PhD contributions, I helped in completing a summary article [LCRT16] about web testing using the state of the art technologies, and I extended the work of my master thesis, that was about the application and validation of a requirements specification refinement method, named DUSM, on a web-based case study, at the end integrated in a journal article [RLRC18]; the method proposed an iterative refinement process of a requirements specification, given in

input, by introducing, for instance, a glossary of terms to reduce ambiguities, inconsistencies and incompleteness, and sketched screen mockups to graphically describe the user interfaces.

In my first PhD year, I proposed an acceptance test driven development approach to generate executable test scripts from user's interactions recorded over HTML stubs of a web application, following a use cases specification as a guideline to describe the interactions, and a capture-replay testing tool to record them [CLRR16a, CLRR16b]. Results from this proposal were promising. Therefore, I investigated further on web testing, trying to overcome the limitations that emerged from the original approach. In particular, I wanted to face and solve the issue of keeping the test artifacts manually aligned to the requirements any time they change (which can happen frequently in the context of the web).

In my second PhD year, I improved and refined the starting idea, proposing a novel approach that integrates UML models to exploit existing transformations tools, to automatically keep requirements specifications and produced test scripts synchronized [CLRR17]. DUSM method was employed to structure and refine the use cases composing the specification, with the aim of making the interactions between the user and the system more explicit, in order to facilitate the next generation of test scripts adhering to the specification.

However, meanwhile the research activity was evolving, a relevant problem emerged. Even if the expressive power of the DUSM method seemed enough suited for the generation of effective test scripts for the web, as experimented in limited case studies, applying the whole approach on real-sized applications was perceived so much time-consuming to doubt the benefits of its adoption. Indeed, specifying a model and precise use cases enriched with screen mockups for a web application to test can be burdensome, even if the initial cost is repaid by a generation of the testing artifacts for free. These unsettling doubts about the approach were eventually corroborated by some reviewers and participants in conferences presentations, who recognized the value of the original proposal, but speculated its infeasibility on real, complex case studies. At that time, I was at the end of my second PhD year.

In parallel to this problem, in 2016, a University project involving some members of the Software Engineering Research Group was launched; the project, entitled *Full Stack Quality for the Internet of Things*, aimed at supporting fresh research ideas concerning the IoT domain. I was involved in a preliminary study conceiving a novel technique for testing IoT systems [LRC+17], then I started growing interest in the opportunities that the IoT field could have provided to my research career. In fact, it emerged that, excluding few vague attempts and white papers, testing IoT systems was mostly overlooked by both academy and industry.

At the same time, a visual tool named Node-RED was released by the IBM labs to support IoT systems development. Node-RED was developed on top of the Node.js framework, implementing the paradigm of flow-based programming. From the beginning, it offered a disparate number of functionalities and services embodied into black-box nodes, to be wired together into flows enabling the communication. Shortly after, Node-RED consolidated itself as a practical solution

to develop IoT systems in a simple manner. The popularity of Node-RED was also fostered by the possibility of easily sharing the own produced nodes and flows among the active community members. Day by day, new nodes and solutions were submitted, and even preliminary testing frameworks were designed. To my eyes, the fact that Node-RED was a visual tool, hence balancing its existence between design and implementation stages, without a support from any existing systematic development and testing approach, was definitely a bonus.

At that point, I consulted my advisors, and together we decided to switch the context of my research from web to IoT, trying to keep unaltered my core research topic, that was still about assuring the quality of systems based on a requirements specification as a backbone.

My final PhD activities then focused on IoT systems development and testing, with a particular dedication to Node-RED. I contributed in formulating an approach for developing and testing IoT systems in Node-RED, starting from a UML model to describe the static and dynamic properties [CLRR18]. In this work, the testing acted on a unit level, exploiting a novel testing framework to test single Node-RED nodes and flows portions. Moreover, to address the rising needs for testing IoT systems as whole beings, considering all the heterogeneous devices they are composed of, an acceptance testing approach of IoT systems was proposed, where a Graphical User Interface (GUI) component is used for the user's interactions [LCO+18, LCF+19]. In this approach, the system behavior was modeled as a UML state machine to describe the changes in the GUI state, and the development involved also Node-RED; the results of its application on a realistic case study were positive. Finally, I completed my PhD activity by proposing a set of guidelines to develop comprehensible Node-RED flows and improve the overall quality, trying to address some common comprehensibility issues that were experienced using Node-RED. An experiment was designed and conducted to evaluate the effect of the guidelines in Node-RED flows comprehension; results showed a significant improvement in the Node-RED flows comprehension, by reducing the number of errors and the time required to complete tasks over some provided Node-RED flows (this work will appear in a conference paper [CLR20]).

In the following, the structure of this thesis document and the origin of its chapters are discussed.

1.1 THESIS OUTLINE AND ORIGIN OF THE CHAPTERS

The thesis is divided into two main parts: *Quality Assurance Approaches for the Web Domain* and *Quality Assurance Approaches for the IoT Domain*. Finally, Appendix *DUSM: A Method for Requirements Specification and Refinement based on Disciplined Use Cases and Screen Mockups* presents a method used to support the activities conducted in the first part, which I applied and validated during my PhD career; to keep this document as cohesive as possible, I chose to report DUSM method in an Appendix, since it does not pertain directly with testing, and thus with the main goal of the thesis.

The works described in the two parts of this thesis address several quality aspects of Web and IoT domains. Although the domains are clearly different, they share the need for a precise requirements specification as a backbone during development and subsequent phases. Indeed, web applications tend to change frequently, making the manual production of aligned test scripts a burdensome activity. Instead, the IoT domain is a fresh field of research and lacks of practical solutions to drive the development and testing activities, where a large number of devices are involved; the addition of supportive tools for the IoT, such as Node-RED, pushes even further the need for a systematic approach.

In both domains, the starting point of the process might be the production of a requirements specification to early detect deviations from the system expected behavior and reduce the manual effort required to produce aligned test cases. In all the proposed works in this thesis, the specification is the core element driving development and testing activities, either given in the form of structured use cases or UML diagrams. Use cases can easily describe the interactions and the order of events occurring between a user and a system, and can drive the production of preliminary but working test scripts. On the other hand, UML can benefit from existing transformation tools to generate valuable contents in a semi-automated way (e.g., abstract paths from a state machine describing the messages exchanged by different devices).

The content of this thesis summarizes and integrates most of the activities conducted during my PhD career, all published in International Conferences and Journals [CLRR16a, CLRR16b, CLRR17, CLRR18, RLRC18, LCO+18, LCF+19, CLR20].

1.1.1 Quality Assurance Approaches for the Web Domain

The first part of this document is structured into four chapters.

Chapter 2 presents a lightweight acceptance test driven development approach for developing web applications, based on a requirements specification provided in the form of use cases and screen mockups (interactive representations of the HTML GUI). A capture-replay testing tool is applied to record interactions performed over the screen mockups, adhering to the descriptions given by the use cases, to generate test scripts able to drive the web application development, using the screen mockups as the baseline. The approach has been applied to re-develop the main features of an open-source web application. The content of this chapter has been published in two conferences: the *16th International Conference on Web Engineering* (ICWE 2016) [CLRR16a] and the *10th International Conference on the Quality of Information and Communications Technology* (QUATIC 2016) [CLRR16b].

Chapter 3 outlines an approach aimed at generating test scripts for web applications from either textual or UML-based requirements specifications. A set of automated transformations are then employed to keep textual and UML-based requirements specifications synchronized and, more importantly, to generate web test scripts from UML artifacts. The transformations are still sketched

and require a proper implementation involving existing transformations tools. The content of this chapter has been published in the *25th International Requirements Engineering Conference Workshops* (REW 2017) [CLRR17].

Chapters 4 and 5, respectively, discuss the related work, conclusions and future work pertaining the research activity conducted on the web domain.

1.1.2 Quality Assurance Approaches for the IoT Domain

The second part of this document is structured into five chapters.

Chapter 6 proposes a set of guidelines to help the Node-RED developers in producing flows that are easy to comprehend and use. The guidelines try to address some comprehensibility issues that may emerge while inspecting and integrating into an existing system a Node-RED flow produced by an external source, or when a system has to move through maintenance activity. An experiment has been conducted to evaluate the effect of the guidelines in Node-RED flows comprehension. Results have shown that the adoption of the guidelines significantly reduces the number of errors and the time required to comprehend Node-RED flows. The content of this chapter has been accepted for publication in the *15th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE 2020) [CLR20].

Chapter 7 presents a preliminary approach for developing and testing a Node-RED system starting from a UML model of its dynamic and static aspects. The elements representing the Node-RED artifacts of the system are generated from the model, along with executable test scripts exercising selected portions of the system behavior. The approach still has to be fine-tuned and implemented in a tool. The content of this chapter has been published in the *1st International Workshop on Ensemble-Based Software Engineering* (EnSEmble 2018) [CLRR18].

Chapter 8 presents an acceptance testing approach of IoT systems adopting graphical user interfaces as the principal way of interaction. In the approach, the development and testing phases are driven by a UML state machine that expresses the expected behavior of the target IoT system. The approach has been applied on a realistic case study, a mobile health IoT system for diabetes management composed of heterogeneous devices, and compared against an existing runtime verification approach, in terms of strengths and weaknesses. Results of the evaluations have shown the effectiveness of the approach, which was able to detect between 71% to 100% of the types of bugs injected in the system. The content of this chapter has been published in the *Journal of Information and Software Technology* (IET Software 2018) [LCO+18] and in the *14th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE 2019) [LCF+19].

Chapters 9 and 10, respectively, discuss the related work, conclusions and future work pertaining the research activity conducted on the IoT domain.

1.1.3 DUSM: A Method for Requirements Specification and Refinement based on Disciplined Use Cases and Screen Mockups

Appendix A presents DUSM (Disciplined Use Cases with Screen Mockups), a method for describing and refining requirements specifications based on disciplined use cases and screen mockups. This method has been adopted as the base to structure, in terms of testing, the requirements specifications treated in Chapter 3. Disciplined use cases are characterized by a quite stringent template to prevent common mistakes, and their descriptions are formulated in a structured natural language, that makes explicit the interactions and the involved data. Screen mockups are precisely associated with the steps of the use cases scenarios to present the corresponding GUI as seen by the human actors before/after the steps executions, improving the comprehension and the expression of the non-functional requirements on the user interface. The method has been validated and evaluated on real case studies and experiments. The content of this Appendix has been published in the *Journal of Computer Science and Technology* (JCST 2018) [RLRC18].

Part II

Quality Assurance Approaches for the Web Domain

Chapter 2

An Acceptance Test Driven Development Approach for Web Applications

Applying Acceptance Test Driven Development (ATDD) in the context of web applications is a difficult task due to the intricateness of existing tools/frameworks and, more in general, of the proposed approaches. The creation of a running test suite before developing a web application is one of the main barriers to the adoption of the ATDD paradigm.

This chapter presents a lightweight approach for developing web applications in ATDD mode, based on the usage of screen mockups (interactive representations of the HTML GUI) and existing web testing tools. The idea, which is the basis of the approach, is using a capture-replay tool to record the interactions performed over the screen mockups created for the web application to develop, and obtain test scripts from such interactions. The test scripts can be directly re-executed on the screen mockups to drive the actual web application development, following its requirements in ATDD mode. The approach has been applied to re-develop the main features of an open-source web application.

The content of this chapter has been published in two conferences: the *16th International Conference on Web Engineering* (ICWE 2016) [CLRR16a] and the *10th International Conference on the Quality of Information and Communications Technology* (QUATIC 2016) [CLRR16b].

2.1 INTRODUCTION

Developing modern web applications is a big challenge for software companies, because they undergo through ultra-rapid development cycles, due to customers' requests and requirements evolution, pushing the release of new features and bug fixes to production in a short time. In

this context, agile approaches and automated testing frameworks are considered among the best choices for web application development and quality assurance [MW01].

The acceptance test driven development is a cornerstone practice [Dow11] that puts acceptance testing and refactoring on top of the software development process. In brief, ATDD is a development process based on short cycles: first, an initial set of failing test scripts is built starting from a feature's specification, usually expressed by means of a user story (i.e., a simple text description consisting of one or more short sentences); then, some code is written to pass each test; finally, some refactoring steps are applied to improve the structural code quality of the web application. However, ATDD is not limited to agile contexts where requirements can be expressed with user stories. Indeed, even more formal requirements specifications based on use cases, such as the one proposed by Reggio *et al.* [RLR15, RRL14], can be used with the ATDD paradigm, where abstract test cases are usually defined by following the scenarios composing the use cases.

A large number of usable *capture-replay* tools (i.e., tools where tester' interactions over a user interface are captured and made re-playable) in the web applications context emerged in the last years. For instance, Fitnium[1], an integration of FitNesse[2] (where test cases can be represented in a tabular form by using the natural language), and Selenium IDE[3] (a browser plug-in that allows to record, edit, and execute web test scripts). Some other tools require a different template for acceptance test cases (like, e.g., the "given-when-then" template of Cucumber[4]) or a freely HTML format which is later enriched by tags to interpret the text and execute it (as for Concordion[5]). Unfortunately, in all these tools, the connections between the test cases and the web application under development (i.e., the so-called fixtures) have to be written by the developer. Indeed, it is well-known that the difficulty of creating a test suite, before the web application exists, prevents the usage of the ATDD paradigm in a real context [HHM11]. Usually, the developers wishing to apply ATDD must manually write complex executable test scripts [BBC10]. This task is cumbersome due to the tight coupling between test scripts and web applications. In fact, in order to work, the test scripts must be able to locate the GUI elements (i.e., web elements) at run-time by using specific *hooks* (for example, identifiers contained in web elements), also called *locators* [LCRS13a, LCRT14], and interact with them. Unfortunately, without having the actual web application it is difficult to foresee such hooks.

Besson *et al.* [BBC10] are among the first researchers to describe an ATDD approach for web applications trying to overcome this burdensome activity. In their work, a web application is modeled with a graph of pages and the paths of the given graph are the test cases, which must be validated by the customer and subsequently transformed into test scripts. Conversely to Besson *et al.*, the approach proposed in this chapter does not require the web application modeling phase,

[1] https://sourceforge.net/projects/fitnium/
[2] http://www.fitnesse.org/
[3] https://selenium.dev/selenium-ide/
[4] https://cucumber.io/
[5] https://concordion.org/

which is substituted by a simpler recording phase of user actions, by means of a capture-replay tool executed upon the previously produced screen mockups.

In order to simplify the adoption of the ATDD paradigm in the web context, this chapter introduces a general lightweight semi-automatic approach that, starting from the textual requirements and the screen mockups of a web application, is able to generate executable functional web test scripts (i.e., black box tests able to validate a web application by testing its functionalities), which in turn drive the development phase. After the implementation of the web application through ATDD, the test scripts are refactored, removing potential reasons of fragility (e.g., hooks that are likely to break during the web application evolution) and code clones, and extended by means of input generator tools, to form a robust regression test suite. This will help developers to produce high quality web applications.

This chapter is structured as follows: Section 2.2 describes the approach, while Section 2.3 shows how the approach is applied to re-develop the main features of the selected open-source web application.

2.2 THE APPROACH

The tasks composing the proposed approach, in the next described, are shown in Figures 2.1 (concerning the development of the web application) and 2.2 (concerning the development of the regression test suite used during the evolution of the target web application).

2.2.1 Requirements Analysis

Requirements analysis aims at producing the requirements specification for the web application under development. It includes the following sub-tasks: (1) eliciting the requirements from future users, customers and other stakeholders, (2) analyzing the requirements to understand whether they are complete, consistent, and unambiguous, and, (3) specifying requirements as use cases or user stories depending on whether, respectively, a more prescriptive or a more agile development approach is adopted.

2.2.2 Mockups Development

Mockups development aims at creating a set of screen mockups used for prototyping the user interface of the web application to develop [HS91, O'D05]. In order to reduce as much as possible the need of manual intervention required to run the automated acceptance test scripts on the web application under development, the mockups have to represent quite accurately – from a

Figure 2.1: From Requirements Analysis to Web Application Development.

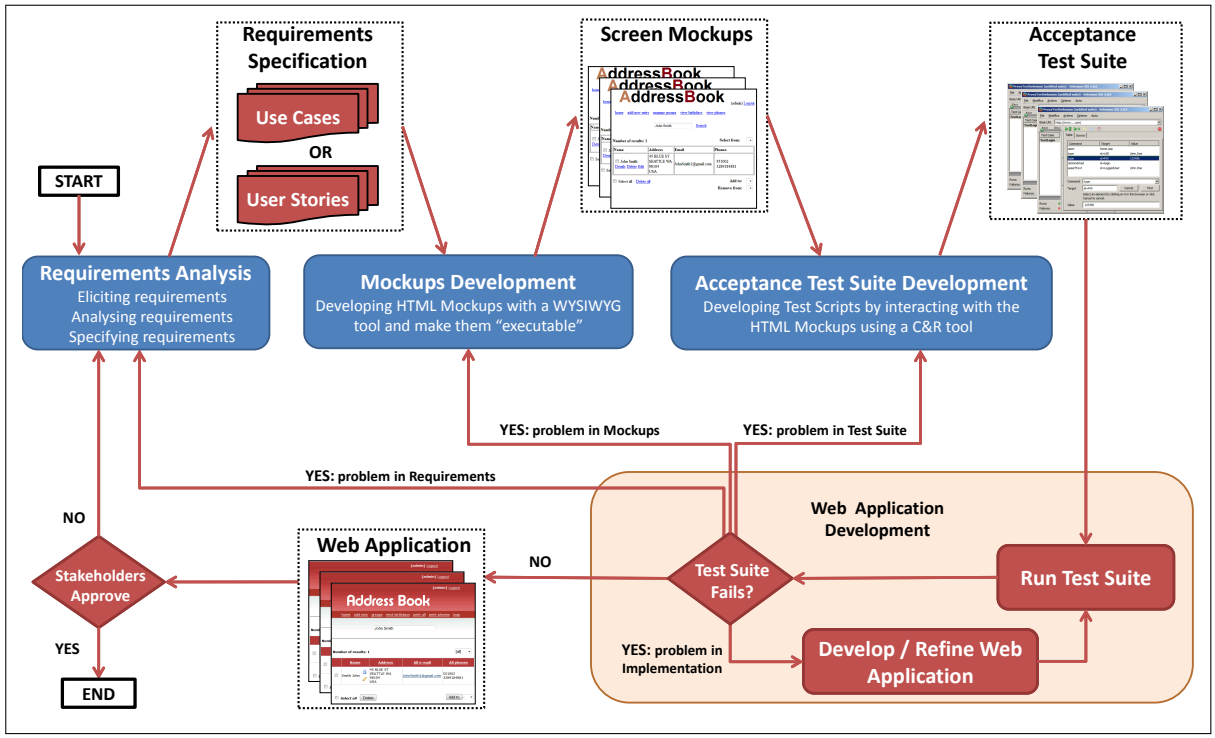
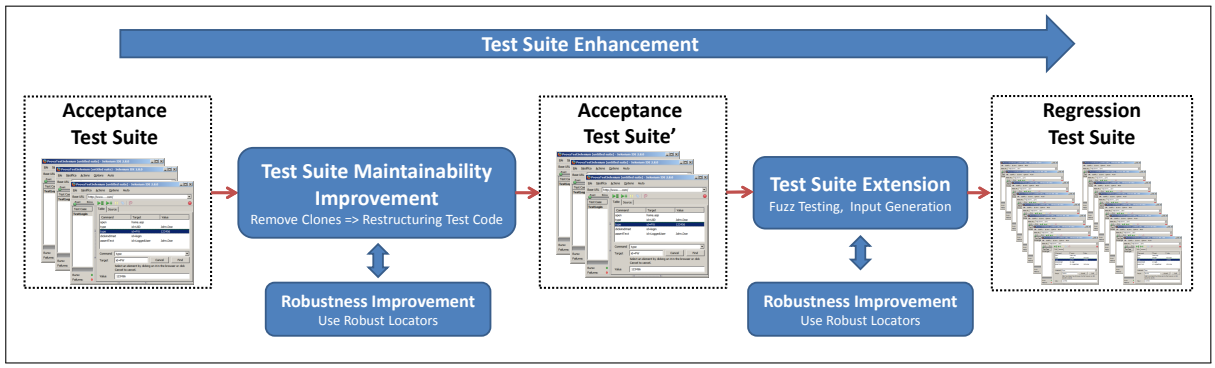


Figure 2.2: Regression Test Suite Development.



functional point of view – the interfaces of the web application (i.e., all the web elements of the web pages to interact with must be shown in the mockups, while the layout and the styles can be just sketched). Since the goal is to use a capture-replay tool (e.g., Selenium IDE) able to generate test scripts on previously created screen mockups of the web application to develop, a WYSIWYG content editor that creates HTML pages represents the best choice to quickly develop them. The

WYSIWYG content editor could be used to specify the properties of the locators that will point to the web elements to interact with. Locators can use many different properties; the most reliable ones are those pertaining identifiers, link textual content, and name values [LSRT16].

2.2.3 Acceptance Test Suite Development

Once the mockups are available, it is possible to record the test suite with any capture-replay tool by interacting with them. To make this task easier and to simulate screen mockups navigability, it is suggested to implement: (1) the links among the mockups and (2) the submission buttons. Concerning submission buttons, it is possible to hard-code the alternative links to different target mockups using JavaScript; for instance, when dealing with a login form we could reach two mockups, "homePage.html" and "wrongPage.html", depending on the inserted values. By hard-coding the alternatives, it would be possible to record the test suite as if it were a real web application. Developing the screen mockups and defining the links among them allows also to produce a preliminary but "working" prototype of the web application that can be shown to the stakeholders. This is very useful for detecting, as soon as possible, problems and misunderstandings in the requirements [RST+14]. More in detail, to record the test suite it is necessary to:

1. Open with the browser the first HTML mockup of a use case/user story and activate the recording functionality of the selected capture-replay tool;
2. Follow the steps described in the use case/user story and replicate them on the HTML mockups (e.g., insert values in the input fields, click links);
3. Manually insert the assertions in the generated test scripts.

Notice that an order of execution of the test scripts must be defined to allow the execution of the entire test suite (e.g., "Delete User Test" must be executed after "Add User Test") and to test each functionality with the corresponding test scripts (e.g., the login functionality must be validated using the "Login Test", thus such test script must be executed before all the others requiring a correct user authentication).

2.2.4 Web Application Development

Web application development is based on a test-first approach using the previously produced test scripts. The functionalities are implemented/refined following the test suite as a guidance until all tests pass successfully. Finally, stakeholders evaluate the resulting web application and decide whether approving it or moving through a further refinement step. It is important to notice that the web application development can be conducted with any technology – e.g., Ajax – and

any development process – e.g., traditional, model-driven (e.g., using WebRatio [ABBB15]), by means of mashups. The only constraint is to use the same text (e.g., for locators based on link textual content) or identifier/name attribute values used in the produced mockups, so that test scripts recorded on the mockups can be executed also on the real web application without any problem.

2.2.5 Test Suite Maintainability Improvement

Once the web application has been developed, the approach moves forward to the test suite maintainability step (see Figure 2.2). Test scripts generated through the recording phase often present repeated instructions (e.g., each time the user has to authenticate herself in the system, the recorded test scripts will include the steps related to the credentials insertion phase) resulting in *code clones*. A good practice is removing code clones by means of refactoring strategies able to encapsulate common test script steps in reusable blocks. After this post-processing step, test scripts are easier to understand and modify and, thus, are more maintainable. In this way, a change in a web application functionality will only impact the reusable blocks instead of propagating the change through the entire test suite. As an example, let us consider the following change in the login page: "for security reasons, provide the password twice instead of once"; without a refactoring step, all the test scripts implementing the login functionality will need a repair. On the contrary, with a refactoring step, only the reusable blocks will need it.

2.2.6 Test Suite Extension

To improve the effectiveness of the test suite and make it more complete, an extension step is needed. Even though test scripts generated as described before can be very useful for developing a web application in ATDD mode, they are not enough to be used for regression purposes, because too simple/limited in terms of code coverage and built using hard-coded values (i.e., the ones recorded during the test script development or contained in the screen mockups). To extend a test suite, at least two categories of tools can be adopted to generate test input data: *input generator tools* and *fuzzers* [BM83]. Inputs generator tools generate input data, often stored in files, to later populate test scripts. On the contrary, fuzzers are tools able to automatically inject semi-random data into a web application, during the exploration. The approach suggests to replace test scripts containing hard-coded values with parametric test scripts, able to read previously automatically generated files containing meaningful input data.

2.2.7 Robustness Improvement

During both the test suite maintainability improvement and extension phases, new interactions with the web page elements can be added. For instance, new assertions can be included to existing test scripts or additional test scripts can require to interact with web elements not considered in the original acceptance test suite (e.g., new values in HTML tables or lists). In these cases, new locators have to be generated. A good practice is making locators robust as much as possible to reduce test suite maintainability efforts. In general, web locators based on identifiers, names and link textual contents are the most reliable, since less prone to change in time [LCRT16]. However, in some cases, web elements cannot be located by any of these properties, and different strategies based on page structure localization must be applied (e.g., XPath). Indeed, if a locator is fragile (e.g., an absolute XPath or an XPath navigating several levels in the Document Object Model of a web page), it will quite surely lead to a test script failure when something changes in the structure of the corresponding web page under test. The *locators fragility problem* can be limited by replacing existing locators with the ones produced by robust locators generator algorithms (e.g., ROBULA+ [LSRT16]). Locators robustness improvement can be performed during both the maintainability improvement and extension phases, as shown in Figure 2.2.

2.3 THE CASE STUDY

The feasibility of the approach has been evaluated by re-implementing an existing web application. As case study, it was chosen the latest version (at the time of writing the papers discussed in this Chapter [CLRR16a, CLRR16b], in 2016, it was version 8.2.5.2) of AddressBook[6], an addresses/phones/birthdays organizer web application already used in several other studies [SM16, HRT16, SLRT16, LSRT15b, LSRT15a, LCRT14]. To make the evaluation more realistic, the approach was applied by separating the roles among the authors of the papers here discussed [CLRR16a, CLRR16b]. More specifically, while Maurizio Leotta was assigned to the requirements analysis phase, I was involved in the subsequent phases. Therefore, from now on, the tasks conducted by Maurizio Leotta will refer to *first author*, while the tasks performed by me will refer to *second author*. For the interested reader, all the produced material (requirements specification, mockups, ATDD test suite, AddressBook prototype, etc.) is downloadable [Cle16].

As described in Figure 2.1, the first step is specifying the requirements for the web application to develop. Thus, for the AddressBook web application, the *first author* performed an exploratory navigation to reverse engineer its most relevant features, that finally were described by means of a requirements specification composed of 15 use cases. The *first author*, playing the role of analyst, adopted DUSM (Disciplined Use Cases with Screen Mockups) [RLRC18], a method for

[6] https://sourceforge.net/projects/php-addressbook/

precisely describing and refining requirements specifications based on disciplined use cases and screen mockups, presented in Appendix A.

In such method, use cases are enriched by a glossary of terms to reduce ambiguity and by screen mockups to better explain what an actor can see before/after each step in a scenario. Use cases follow a quite stringent template, and must adhere to a set of constraints for the whole artifacts of the specification (i.e., use cases description, glossary, screen mockups) to improve their consistency. This method was chosen since it helps in better describing scenarios and making performed interactions more explicit and clearer.

Moving forward with the process, the *second author* selected BlueGriffon[7] tool as the WYSIWYG content editor to design the AddressBook screen mockups, since it presents a simple interface and provides HTML pages to be later used to record test scripts. For each screen mockup to represent, only a subset of the original web elements was reproduced, filtering out those that were not interesting, not useful or not clear (e.g., the web application at hand presents many links that perform similar tasks). Reusable web elements were identified and shared in every screen mockup (e.g., a link to home page).

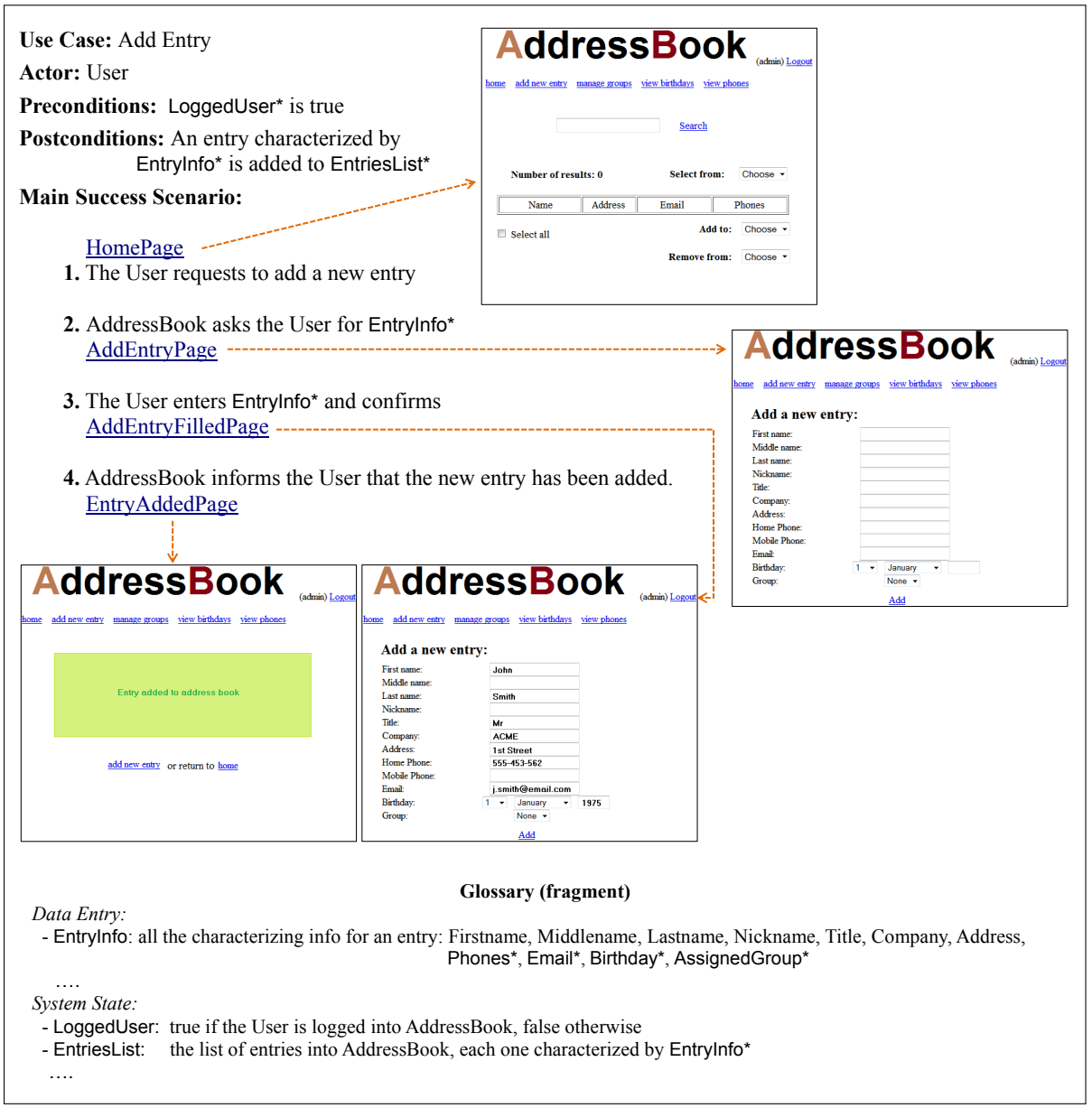
Screen mockups were created by following the order suggested by use cases scenarios, together with some guidelines from the adopted method [RLRC18] which helped in the process. In the case of AddressBook, the login page mockup was the first one, followed by the home page mockup and so on, in accordance with the use cases functionalities previously captured. Figure 2.3 shows a full example, where four generated screen mockups are linked to a use case. The screen mockups were linked to use cases steps any time the system had to show/ask something to the user (e.g., a message or an empty form) or the user had to provide some data (e.g., filling a previously shown empty form). The use case given in Figure 2.3 depicts the screen mockups as hyperlinks to the actual HTML representations produced with BlueGriffon; for example, the *AddEntryPage* hyperlink after step two is a reference to the screen mockup pointed by the arrow. As suggested by DUSM method [RLRC18], a glossary of terms was introduced to reduce the overall ambiguity; the terms followed by the star symbol in Figure 2.3 (e.g., LoggedUser*) are references to entries in the glossary. For the sake of brevity, in Figure 2.3 just a fragment of the glossary is shown.

As suggested by the approach, links among screen mockups were implemented and, when necessary, manually injected with Javascript code to handle input alternatives (e.g., correct/wrong credentials) in order to make them executable.

Overall, 35 screen mockups were produced, 18 of them uniquely representing the main sections of the application and the remaining ones derived from the former. Since screen mockups are static items, the derived ones were necessary to simulate the behavioral aspect of the web application to develop (i.e., its states). For example, the home page should list all the entries in the system; therefore, any time data were added/removed/updated, a new screen mockup was created to represent the change.

[7] http://www.bluegriffon.org/

Figure 2.3: Add Entry use case, adorned with screen mockups and a glossary fragment.



In order to guarantee a correct link between the generated test scripts and the future web application, the *second author* assigned identifiers or names as locators to the web elements contained in the HTML screen mockups on which interactions will occur (e.g., a text field that will be filled) and to those that will be likely involved in assertions on data containers (e.g., a label that will show an output message consequent to some user actions).

For the test suite development, the *second author* selected Selenium IDE capture-replay tool, a browser plug-in that allows to record, edit, and execute web test scripts. Selenium IDE was chosen among the others capture-replay tools since it is largely used [LCRT16], it has a simple interface, and offers a large variety of useful extensions. Adopting a capture-replay tool like Selenium IDE allows to pay little attention to the mockups graphical aspect and focus on the user interaction, with clear advantages in terms of effort required for creating the test scripts [LCRT13].

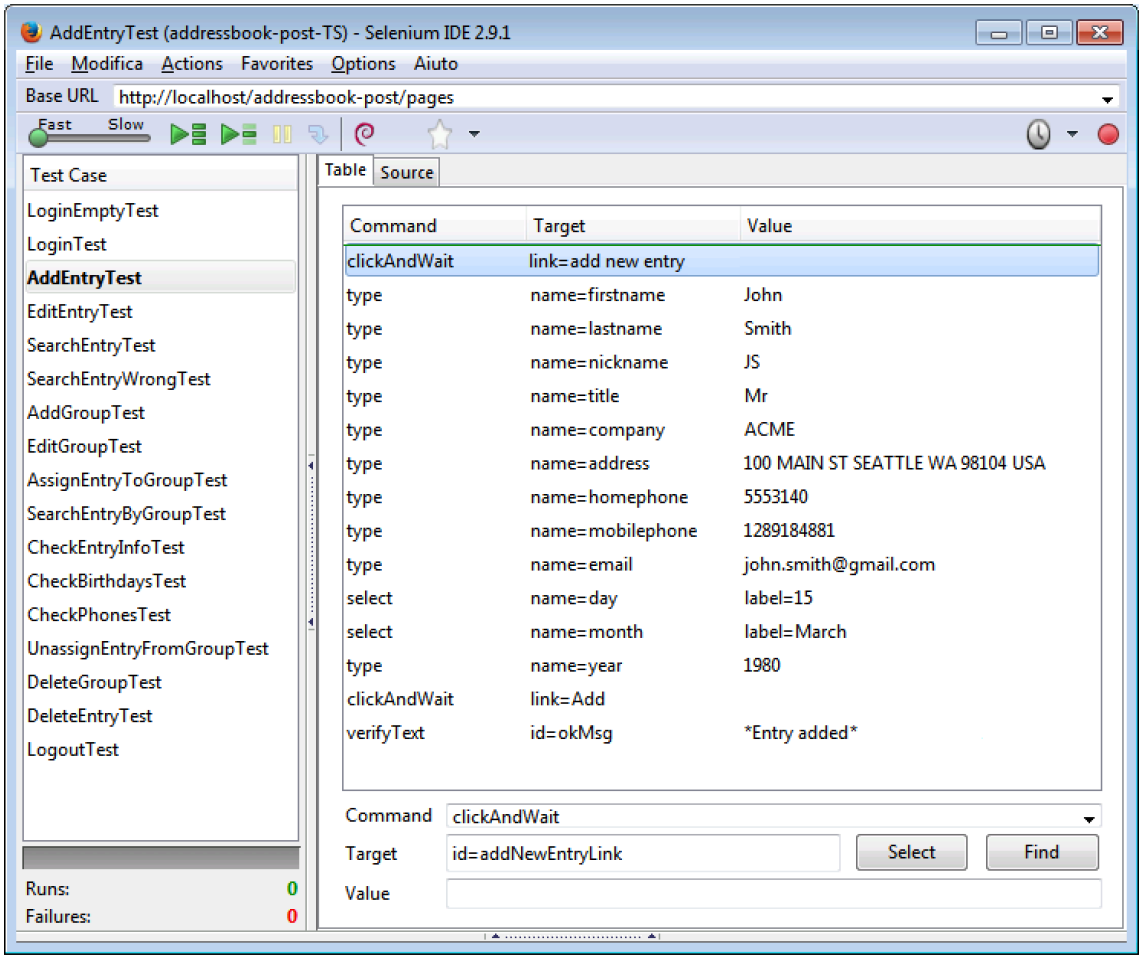
The presence of both use cases and screen mockups allowed the *second author* to record test scripts during the process, since interactions naturally followed the use cases scenarios. Moreover, screen mockups enriched the comprehension of which interactions and data were needed. For example, *AddEntryPage* and *AddEntryFilledPage* hyperlinks from Figure 2.3 refer to screen mockups that represent a form-based web page, before and after filling it with data, so they suggest which input the tester may use to fill the text fields with.

Each test script was generated starting from use cases scenarios. Initially, the *second author* had to open the first screen mockup of each use case, according to the listed pre-conditions. For instance, if a pre-condition states that a list of entries is shown, then the test script should start from the screen mockup where that list is actually displayed. Then, the recorded interactions on the web elements followed the scenarios steps, while assertions were manually added, guided by the post-conditions described in the use cases. As an example, if a post-condition states that an entry is added to a list, then that entry should be visible in the list and the system should inform about the completion of the operation (to be checked by means of an assertion).

The final acceptance test suite was composed of 17 test scripts. A sample one, expressed in the *Selenese* language, is shown in Figure 2.4. Selenese is the script language used by Selenium IDE; each Selenese line is a triple having form *(command, target, value)*, where the *command* determines the action performed over an element, the *target* uses a locator to identify the element within the web page, and the *value* is the optional input used for certain kind of actions (e.g., to fill a text field). In Figure 2.4, the test script clicks on the link identified by the "add new entry" textual locator (more in detail, the text of a link) to access the page where the entry can be added. Then, it performs some interactions on form fields located by the name property (i.e., text fields and drop down menu), to enter/select input data for the new entry, and confirms the insertion by clicking on the "Add" link. Finally, the assertion is manually added and checks whether the confirmation message is shown in a label located by a specific identifier (i.e., *okMsg*).

The previously generated screen mockups representing unique aspects of AddressBook (18 out of 35) were adopted to re-develop the web application (17 mockups were excluded from the process because they were just mere instantiations, i.e., replications with different data). According to the ATDD paradigm, development was guided by test scripts, step by step. As expected, they failed at the first run, so, in order to pass the tests and fulfill the expected features, the *second author* had to implement AddressBook dynamic behaviors (mostly through PHP code to handle actual data and navigation). The test scripts failures were easy to detect and fix since Selenium IDE provides quite explicative messages. Following the process, the *second author* had to switch back

Figure 2.4: Add Entry test script in Selenium IDE tool.



to mockups and test suite development just a couple of times, due to minor changes in the GUI or in interactions.

At the end, the recorded test suite led successfully to a preliminary but working AddressBook web application. Notice that no changes to the web elements were needed, since the same locators of the screen mockups were preserved. From this point, the GUI layout could be enriched with no impact on test scripts.

The *second author* then proceeded to enhance the test suite, as shown in Figure 2.2. To factorize test scripts, he used the Selenium IDE *rollup* command to group repeated sequences (i.e., clones) of Selenese instructions and reuse them across different test scripts in the test suite. The rollup command refers to a Javascript file that stores the shared Selenese triples (i.e., command, target, value) as a set of rules to be called any time they must be executed. In Figure 2.5 a Javascript

Figure 2.5: Add Entry Javascript rollup rule.

```
var manager = new RollupManager();
manager.addRollupRule({
    name: 'add_entry',
    description: 'add a new entry',
    args: [],
    commandMatchers: [],
    getExpandedCommands: function(args) {
    var commands = [];
    commands.push({command:'clickAndWait', target:'link=add new entry'});
    commands.push({command:'type', target:'name=firstname', value:${firstname}});
    commands.push({command:'type', target:'name=lastname', value:${lastname}});
    commands.push({command:'type', target:'name=nickname', value:${nickname}});
    commands.push({command:'type', target:'name=title', value:${title}});
    commands.push({command:'type', target:'name=company', value:${company}});
    commands.push({command:'type', target:'name=address', value:${address}});
    commands.push({command:'type', target:'name=homephone', value:${homephone}});
    commands.push({command:'type', target:'name=mobilephone', value:${mobilephone}});
    commands.push({command:'type', target:'name=email', value:${email}});
    commands.push({command:'select', target:'name=day', value:${day}});
    commands.push({command:'select', target:'name=month', value:${month}});
    commands.push({command:'type', target:'name=year', value:${year}});
    commands.push({command:'clickAndWait', target:'link=Add'});
    commands.push({command:'verifyText', target:'id=okMsg', value:'Entry Added'});
    return commands;
}});
```

rollup rule is shown. It includes all the interactions with the web application needed to add a new entry (i.e., clicking the link to visit the page where the form is located, inserting the data into the form, and confirming it, as shown also in test script of Figure 2.4). Thus, this rule can be referred inside Selenium test scripts through the specified property name (i.e., *add_entry* on top of Figure 2.5) in place of the Selenese triples. In this way, interactions upon highly used web elements, such as text fields inside a login form, are contained in the file, therefore any change that may influence those elements will impact just the rule, and eventually test scripts become more robust and readable.

The adoption of rollup rules can provide substantial advantages for what concerns the effort of maintaining the test suites. For instance, Leotta *et al.* [LCRT16] show the benefits of adopting the *Page Object* pattern[8], a quite popular web test design pattern, which aims at improving the test case maintainability and at reducing the duplication of code. A Page Object is a class that represents the web page elements and that encapsulates the features of the web page into methods. With the Page Object pattern, each method can be reused more times in a test suite. Thus, a change at the level of the Page Object can repair more than one test case at once. From the maintainability point of view, having rollup rules or Page Object methods is quite similar. In total, 14 rollup rules were defined in AddressBook test suite, saving 51 lines of code (LOCs) due to repeated instructions. Clearly, the benefits of using rollup rules depend mostly on the test suite size and

[8] http://martinfowler.com/bliki/PageObject.html

Figure 2.6: Add Entry XML data file structure.

```
<testdata>

      <vars  firstname   = "..."
             lastname    = "..."
             nickname    = "..."
             title       = "..."
             company     = "..."
             address     = "..."
             homephone   = "..."
             mobilephone = "..."
             email       = "..."
             day         = "..."
             month       = "..."
             year        = "..."
      />

</testdata>
```

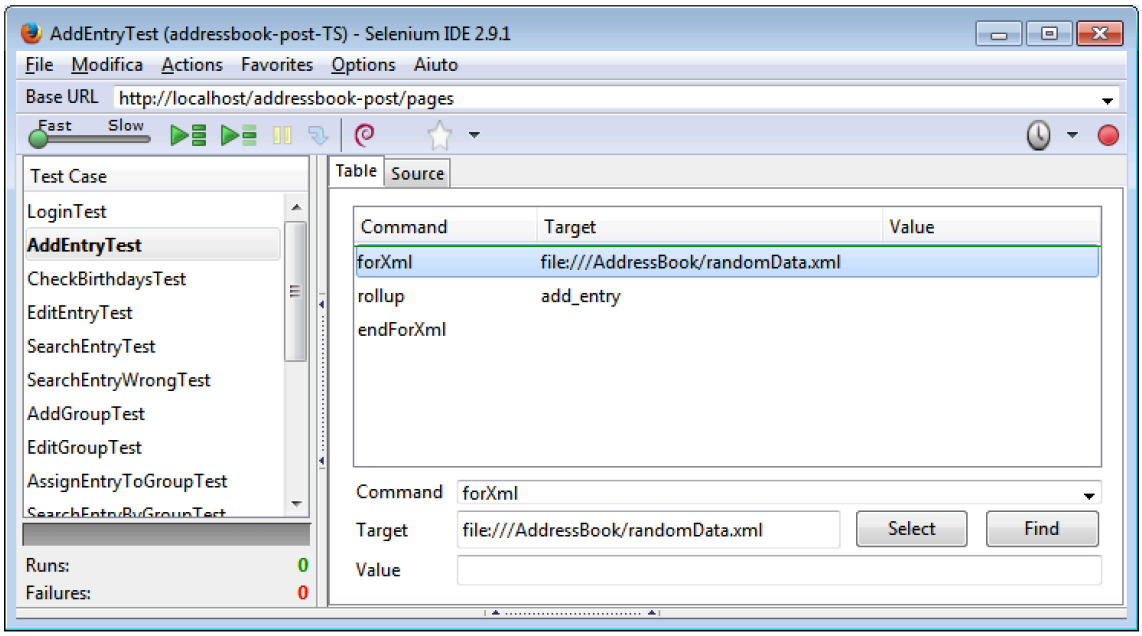
on the number of times a block of instructions is reused among the test scripts (e.g., adding a new user). This becomes particularly evident in the next phase of the approach (i.e., Test Suite Extension, see Figure 2.2), where the test scripts are re-executed several times with different data.

Since generated acceptance test scripts contained hard-coded input data by construction (see Figure 2.4), the *second author* transformed them in parametric test scripts, relying on XML datasets created by the GenerateData[9] input generator, and executed them with SelBlocks[10], a Selenium IDE extension able to execute parametric test scripts. Even though the generated data were unable to cover all possible scenarios which may originate from feeding a form with input data, it was enough to drastically augment the test suite produced in the previous phase of the approach. Also, the old hard-coded test scripts were kept, since they were still useful for testing purposes (the meaningful data recorded are able to cover, at least once, each existing scenario). To make multiple data understandable by SelBlocks, just few changes in their XML structure were needed; in Figure 2.6 the accepted test data template is shown, where the *vars* tag represents a random entry instance with its attributes.

Furthermore, test scripts were enriched by additional instructions, such as loops that cycle across the XML entries or parameterized commands that take in consideration the multiplicity of the given data (e.g., clicking on the exact edit link associated to the currently selected user). Consequently, parameterized assertions were manually introduced as well. At the end, the test suite which was originally composed of 17 hard-coded test scripts was augmented to 487 executable test scripts, since some of the original ones were parameterized with several different input/expected values previously stored in XML datasets. In Figure 2.7, the enhanced version of the test script shown

[9] http://www.generatedata.com/
[10] https://github.com/refactoror/SelBlocks/

Figure 2.7: Add Entry *enhanced* test script in Selenium IDE tool.



in Figure 2.4 is given. The test script is now evidently shorter, since all the instructions to add a new entry are enclosed in the rollup command (Figure 2.5), while the *forXml* and *endForXml* commands are used to iterate across the provided XML file.

Finally, the *second author* applied the ROBULA+ algorithm [LSRT16] to generate robust locators for the new web elements used in the additional test scripts and for the further assertions added to the existing ones, in all cases in which Selenium IDE relied on fragile navigational XPath locators. ROBULA+ follows a top-down approach that takes in input an absolute XPath expression and specializes it by iteratively applying 7 transformations to the head of the expression, in order to produce relative and more robust alternatives. These candidate locators can be generated, for instance, by adding to the expression an attribute or a set of attributes extracted from the DOM element pointed by the original locator, by replacing the '*' symbol in the path with the existing ancestor tag, and by adding tags levels. The transformations are applied until a unique locator is found or an absolute XPath expression similar to the one taken in input is generated. By using ROBULA+, it was possible to improve test scripts robustness by means of more robust XPath locators, which is particularly useful in case of dynamic DOM structures, such as tables, that contain changing data that cannot be easily provided of meaningful locators.

2.3.1 Limitations and Improvements

The approach is feasible and simple to apply for small and medium sized web applications, but probably the same process would be quite costly for big web applications. In such cases, it may lead to a high number of screen mockups, and so to a relevant effort from the mockups designers point of view. Moreover, to limit the number of screen mockups to be produced, interactions upon web elements must be reduced as well. Unfortunately, this has the consequence to limit the number of functionalities tested and the coverage of the test scripts produced. Second, to benefit from the approach and keep test scripts runnable, mockups designers must preemptively associate robust locators, such as meaningful identifiers or names, to web elements (even to labels that may potentially express some useful data to assert). This can be a cumbersome task. Third, reproducing the navigation among mockups and injecting Javascript code to simulate dynamic behaviors can also be a tedious and time-consuming task, if manually performed. Moreover, test suite enhancement requires additional Javascript code to introduce rollup rules, while the generated input data needs some improvements to affect more scenarios and provide smarter datasets.

To limit the cost of manually producing test scripts (i.e., via capture-replay tools) and to keep them synchronized to the requirements specification, that could frequently change in the dynamic context of the web, the approach should be supported by a model of the web application requirements specification. The model should precisely describe the requirements, and, based on a notation (e.g., UML), be able to exploit existing state of the art technologies to automatically regenerate the test scripts from the model, once the requirements change.

Some of these limitations are addressed in Chapter 3.

Chapter 3

Generation of Web Test Scripts from Textual or UML-based Specifications

Web applications pervade our life, being crucial for a multitude of economic, social and educational activities. For this reason, their quality has become a top-priority problem. End-to-End testing aims at improving the quality of a web application by exercising it as a whole, and by adopting its requirements specification as a reference for the expected behavior.

This chapter outlines a novel approach aimed at generating test scripts for web applications from either textual or UML-based requirements specifications. A set of automated transformations are employed to keep textual and UML-based requirements specifications synchronized and, more importantly, to generate End-to-End test scripts from UML artifacts.

The content of this chapter has been published in the *25th International Requirements Engineering Conference Workshops* (REW 2017) [CLRR17].

3.1 INTRODUCTION

In the last years, web-based software has become the key asset in a multitude of everyday activities. For this reason, effective testing approaches aimed at increasing the quality of web applications are of fundamental importance. End-to-End testing is a relevant approach for improving the quality of complex web systems [LCRT16]: web applications are exercised as a whole, testing the full-stack of technologies implementing them. It is a type of black box testing based on the concept of test scenario, i.e., a sequence of steps/actions performed on the web application (e.g., insert username and password, click the login button, etc.).

A requirements specification expressed as use cases can be employed as a reference for the correct behavior of a web application, and can be used to derive the test cases. Screen mockups are additional artifacts used in conjunction with use cases to represent the interface of a web application before/after the execution of each scenario step; they can improve the comprehension of functional requirements, and can also be used for the non-functional ones [RLRC18]. Moreover, the introduction of a glossary, to precisely describe the terminology referred by use cases, can enforce the requirements specification understandability, reducing also ambiguities which may originate from unclear or complex sentences. A method providing well-formedness constraints over such entities would thus produce a precise and of high quality requirements specification [RLRC18], also in a highly dynamic context as the Web. Indeed, use cases plus screen mockups naturally describe how a web application should be tested in terms of its behavior, as perceived by the users, and the glossary may clarify the data used by test cases, as well as the performed instructions.

Despite their wide adoption for describing requirements, textual use cases and, more generally, natural language processing techniques, cannot directly support automated test cases generation [GEL+16]; instead, different notations (e.g., UML) may provide a more structured and formal view, exploitable by existing tools. For example, state machines integrated with screen mockups can intuitively represent the system behavior as navigational paths, basis for future test cases generation.

This chapter outlines an approach for generating End-to-End test scripts for web applications from a *precise* requirements specification, either textual or UML-based. The precise requirements specification satisfies a set of well-formedness constraints aimed at improving the overall quality and making the specification suitable for test scripts generation. Precision can also clarify how the functionalities of the web application have to be developed and tested, hence a specification compliant to such rules may work as a reference manual. To generate a specification, the analyst may choose the perspective (s)he is more confident with (i.e., textual or UML) and, by means of an automated transformation, derive the other one with a little effort. An additional automated transformation is applied on the UML artifacts to generate End-to-End test scripts. The approach is currently tailored to the generation of test scripts for web applications, but with some adjustments it could be used to test also different kinds of systems (e.g., mobile apps). The approach is based on DUSM (Disciplined Use Cases with Screen Mockups) [RLRC18], a method for precisely describing and refining requirements specifications based on disciplined use cases and screen mockups, presented in Appendix A.

Even though the generation of UML artifacts from use cases and consequent test cases extraction has been already investigated (for example, [YAB11, KR03, SMB08]), the approach outlined in this chapter aims at generating End-to-End test scripts for web applications, completely aligned with their requirements specifications, where textual and UML-based specifications are kept synchronized by means of automated transformations, and screen mockups are integrated in the process and are functionally complete to be exercised by the test scripts. The approach is intended

to be supported by a prototype tool, to assist the final user in the definition of the requirements specifications, and in the automated artifacts generation (e.g., UML diagrams, test scripts). The tool will provide a user-friendly and step-by-step interface, where manual intervention is reduced as much as possible.

In the following, the approach is described. Although it does not specifically address the ATDD approach introduced in Chapter 2, the adoption of a precise model to describe a web application requirements specification, and generate from it all the test artifacts, may overcome some of its limitations, in particular the manual effort required to obtain and maintain effective test scripts via capture-replay tools.

Section 3.2 of this chapter provides an overview of the approach, Sections 3.3 and 3.4 describe the textual and the UML-based requirements specifications, respectively, and finally the transformations between the specifications and into the testware are discussed in Sections 3.5-3.6.

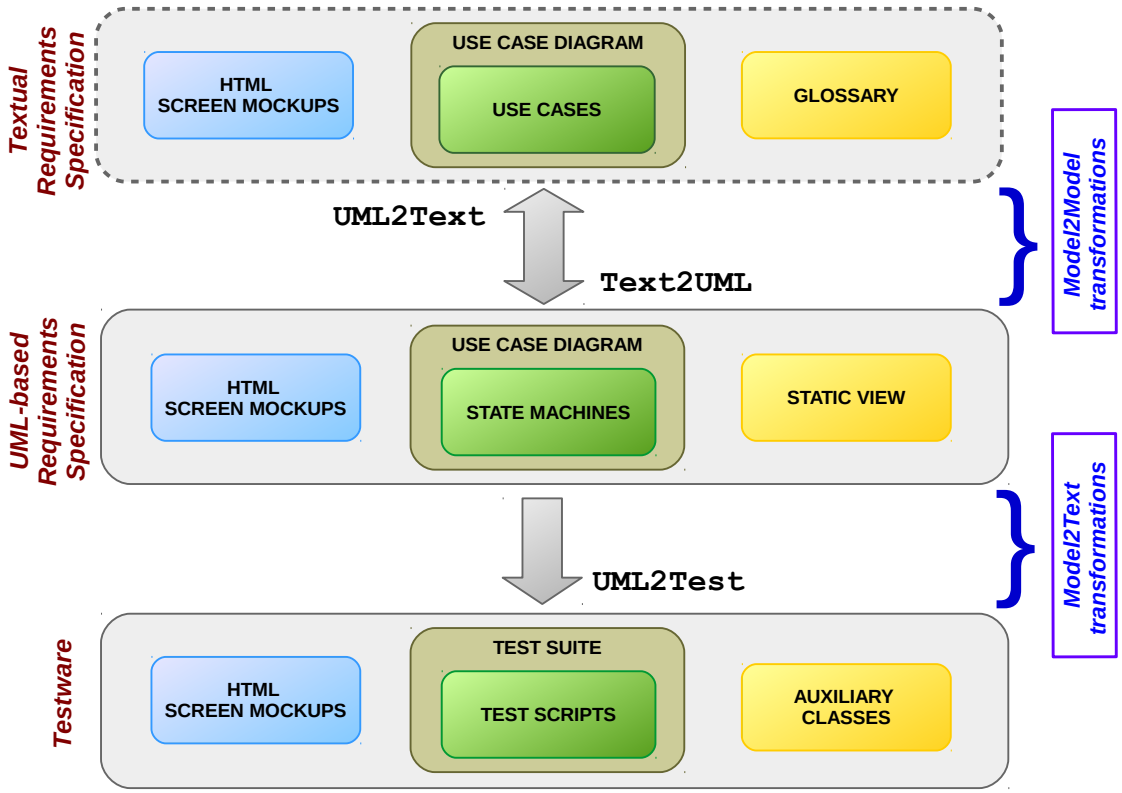
3.2 THE APPROACH

The aim of the approach is to generate test scripts for a web application given its requirements specification in input. Figure 3.1 provides an overview of the approach, which is based on two kinds of requirements specifications, *Textual* and *UML-based*, and three main automated transformations (**`Text2UML`**, **`UML2Text`**, **`UML2Test`**), in the following discussed.

The *Textual Requirements Specification* is usually the starting point of the approach, and is characterized by a use case diagram, textual use cases, HTML screen mockups, and a glossary introducing the used terminology. Use cases are adopted since they naturally represent the behavior of a system as structured scenarios. Screen mockups are embodied in use cases steps to enhance the overall comprehension of the requirements [RLRC18], and to support the ensuing automated test generation and execution [CLRR16b]; in fact, they visually describe how the web application GUI should appear, and how it should react to users' interactions [RLRC18]. The textual requirements specification, discussed in the following of this chapter, is based on DUSM methodology, presented in Appendix A.

The *UML-based Requirements Specification* is characterized by a use case diagram, use cases given in the form of UML state machines with attached screen mockups, and a static view (i.e., a UML class diagram) defining the used data, the operations over them, and the interactions between the actors and the web application. Among the variety of dynamic UML diagrams, state machines are chosen to represent use cases, since they have the dual benefit of concisely describing the interactions occurring between the application and the user and are able to naturally represent the GUI changes based on the external events, whereas activity and sequence diagrams are more focused on representing the ordered sequences of actions performed by the user and often involve

Figure 3.1: An overview of the approach.



a larger number of constructs, which may turn impractical to use for long scenarios and result more complex to understand.

Each kind of specification, either textual or UML-based, is defined by means of a meta-model accompanied by a set of well-formedness constraints. This to ensure the effectiveness of the transformations between the two kinds of specifications.

Finally, the *Testware* is the output of the approach; it includes the test suite, grouping the automatically generated test scripts that cover all the interesting aspects described in the requirements specification, the screen mockups over which the test scripts instructions are performed, and the auxiliary classes coding the data, the operations over them, and the occurring interactions.

The **`Text2UML`** and **`UML2Text`** transformations aim at moving between the textual and the UML-based requirements specifications. More specifically, **`Text2UML`** transforms use cases into state machines, and the glossary into the static view. Instead, **`UML2Text`** generates use cases from state machines, and the glossary from the static view. Notice that the screen mockups and the use case diagram are untouched by the transformations; in fact, they are complementary elements

in both kinds of requirements specifications. **Text2UML** and **UML2Text** are *Model2Model*[1] transformations, since they rely on the meta-models defining the form of the textual and the UML-based requirements specifications. Finally, **UML2Test** is a *Model2Text*[2] transformation generating the code of the testware from the UML artifacts.

The approach is applicable from scratch, integrating it within an ATDD methodology to drive the development of a novel application, as the approach introduced in Chapter 2, where screen mockups are employed as the basis for web pages development [CLRR16b], as well as to test already existing web applications, having at hand a precise form of their requirements specifications.

The approach allows to skip a textual formulation of the requirements (that is the reason of the surrounding dashed line in Figure 3.1, indicating optionality), starting directly from UML, from which a textual counterpart can be automatically derived. Having two different perspectives gives more freedom to the analyst, who may alternatively choose a simpler textual solution to be transformed into UML models or directly adopt UML in case of high professional skills. However, the testware is generated from UML only, as shown in Figure 3.1, since UML represents use cases in a more structured and formal way and can be exploited by existing model transformation tools.

From now on in the chapter, the term WebApp is used to denote a generic web application to test, after having specified its requirements, while the running example chosen to present the approach will be referred as PhoneBook, a simple web application storing phone contacts info, which basically corresponds to the AddressBook case study of Chapter 2, with some new features and GUI adjustments. The complete PhoneBook textual and UML-based requirements specifications can be found in [CLRR].

3.3 TEXTUAL REQUIREMENTS SPECIFICATION

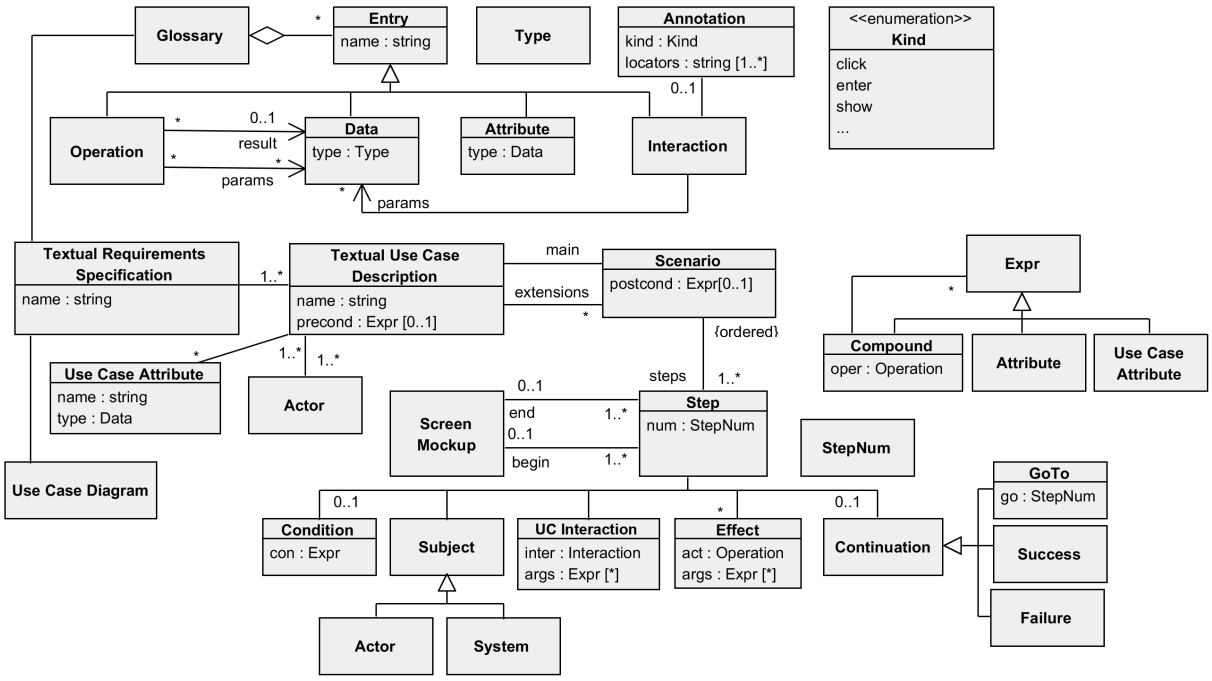
The textual requirements specification consists of a UML use case diagram summarizing the use cases, a glossary that lists and makes precise all the terms used in the use cases, a description of each use case, and a set of HTML screen mockups associated with use cases steps. It is precisely defined by the meta-model of Figure 3.2, and is based on DUSM [RLRC18] method, presented in Appendix A.

3.3.1 Use Case Diagram

The *use case diagram* summarizes the WebApp use cases, making clear the actors (i.e., the users of the WebApp) taking part in them, and their mutual relationships. It is actually a UML diagram,

[1] https://projects.eclipse.org/projects/modeling.mmt/
[2] https://projects.eclipse.org/projects/modeling.m2t/

Figure 3.2: The textual requirements specification meta-model.



but quite simple to understand and to produce, and useful to summarize the use cases, so there is no need of a more detailed presentation.

3.3.2 Glossary

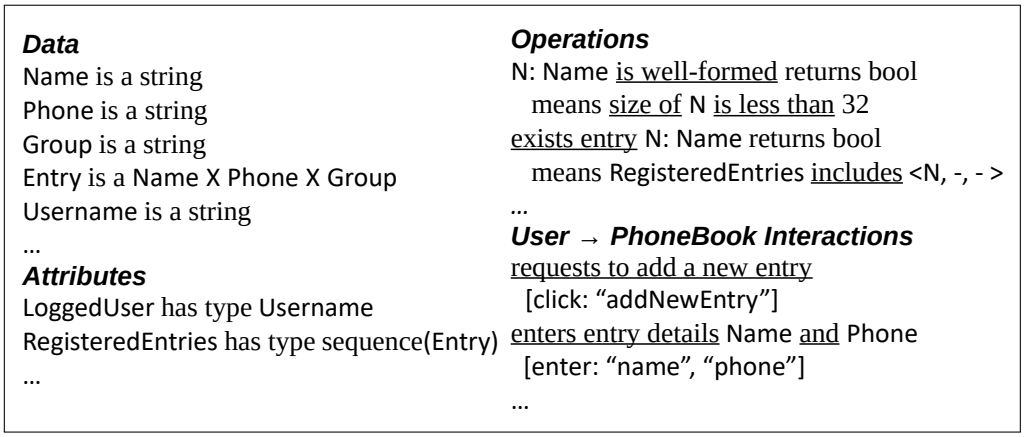
The *glossary* is a list of entries, introducing all the terms appearing in the use cases, each one consisting of a name, and of a definition. The glossary entries are distinguished in: **Data**, **Attributes**, **Operations**, and **Interactions**. A portion of the PhoneBook glossary is shown in Figure 3.3.

Data: the types of the data mentioned in the use cases. They have form "name is a *type*", where *type* is either a basic type (e.g., string, int, bool), a Cartesian product, or a sequence of types. For example, in Figure 3.3 we have "Name is a string", and "Entry is a Name X Phone X Group".

Attributes: the properties abstractly describing the updatable state of the WebApp. They have form "name has type *Data*", where *Data* is a (sequence of a) previously defined data. For example, in Figure 3.3 we have "LoggedUser has type Username" and "RegisteredEntries has type sequence(Entry)".

Operations: the functions performed over data and attributes to set/get their contents. They have form "$\underline{\text{namepart}_1}$ Data_1 ... $\underline{\text{namepart}_n}$ Data_n [returns Data]", where each Data_i is a previously defined data and the operation may have or not a return type. The semantics of the operations is

Figure 3.3: A portion of PhoneBook glossary.



given in a structured textual form. For example, in Figure 3.3 we have "N: Name is well formed returns bool", which checks that the size of a Name N is less than 32 characters, where "size of" and "is less than" are predefined functions over strings and integers.

Interactions: the atomic interactions between the actors and the WebApp and vice versa. They have form "namepart$_1$ Data$_1$ ... namepart$_n$ Data$_n$", where each Data$_i$ has been previously defined. For example, in Figure 3.3 we have requests to add a new entry, and enters entry details Name and Phone. The parts enclosed by square brackets are the *annotations* of the interactions (see the corresponding Annotation class in the meta-model of Figure 3.2); each annotation determines the *kind* of the interaction (e.g., click on something, enter some data, and so on) and the web element(s) *locator(s)* on which the interaction is performed, as further explained in Section 3.3.4.

3.3.3 Use Cases Descriptions

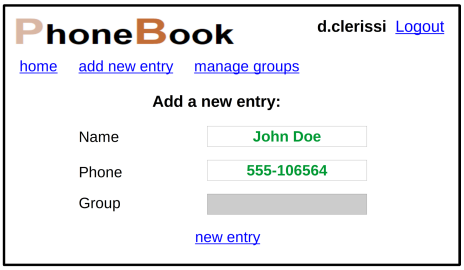
In the approach, use cases follow the template proposed in DUSM method [RLRC18]. An example of a PhoneBook use case is shown in Figure 3.4.

The *use case attributes* are the data needed to describe the use case, each one characterized by a name and a type taken from the data introduced in the glossary. In Figure 3.4, two attributes are declared: Name N and Phone P, where Name and Phone are data defined in the glossary of Figure 3.3.

The *preconditions* state what we assume about the current state of the WebApp before the execution of the associated use case (optional). They are expressions built using the data, the attributes and the operations defined in the glossary, and are shared among all the scenarios composing the use case description. In Figure 3.4, a precondition concerning the authentication of the user is introduced.

Figure 3.4: PhoneBook Add Entry use case.

Use Case: Add Entry
Actors: User
Attributes: Name **N**, Phone **P**
Preconditions: LoggedUser is defined
Main Success Scenario:
1. User requests to add a new entry Homepage
2. PhoneBook asks for entry details. AddEntryPage
3. User enters entry details **N** and **P**. AddEntryPage
4. User confirms entry details. AddEntryPage
5. If **N** is well-formed and not exists entry **N**, then PhoneBook shows entry added;
<N, P, none **>** is added to RegisteredEntries. ValidEntryPage



Extensions:
5a.1 If not **N** is well-formed, then PhoneBook shows invalid name. InvalidEntryPage
5b.1 If exists entry **N**, then PhoneBook shows entry already exists. InvalidEntryPage

The *main success scenario* describes the basic execution of the use case, whereas the *extensions* (any number, also none) define all the other possible executions. Scenarios are sequences of uniquely numbered and ordered steps, each one structured as the following, where square brackets indicate optionality:

[if *Condition*, then] *Subject Interaction*; *Effect**. [*Continuation*.]

The *Condition* determines the step executability and is formulated as the previously described preconditions; for example, in Figure 3.4 a condition is associated with step 5 and checks if the entry name is well-formed and not already present in the registered entries (the definitions of these operations are given in the glossary, see Figure 3.3). The *Subject* of a step is either an actor or the WebApp, while the *Interaction* is a sentence describing either what flows from the actor towards the WebApp or vice versa; interactions must be formulated by using those listed in the glossary, like the underlined sentences in the use case steps of Figure 3.4. The *Effect* of a step is a sequence of sentences written by using the operations (having a side effect) listed in the glossary, describing how the WebApp state changes depending on the *Interaction*; for example, at the end of step 5 in Figure 3.4, a new entry is added to the registered ones. Finally, the *Continuation* defines how the use case flow continues after the end of the step; it may be a jump to a step different from the following one (GoTo class in Figure 3.2) or a sentence declaring the success or the failure of the use case execution. If there is no *Continuation*, it means that the flow continues to the following step.

Successful scenarios can optionally be associated with *postconditions*, as shown in the meta-model of Figure 3.2, which are expressions built similarly to preconditions, but specific for a completed scenario.

3.3.4 Screen Mockups

The *screen mockups* are GUI sketches representing accurately - from a functional point of view - the interfaces of a WebApp. As discussed in Chapter 2, since the considered domain is the web, screen mockups are HTML pages expressing a WebApp GUI in terms of its interactive web elements. For each use case step, a begin and an end screen mockup can be linked as placeholders [RLRC18] (i.e., hyperlinks to the corresponding HTML pages) to represent how the GUI looks before and after the step execution. At least one mockup is needed for each step, since a step describes some interactions performed over the web elements. For example, in Figure 3.4, a screen mockup is linked at the end of each step.

Any screen mockup associated with a step must be consistent with it, i.e., it should present the same informative content, otherwise the introduction of the mockup would be the cause of ambiguities in the requirements specifications, instead of improving their quality [RLR15]. The consistency between mockups and use cases steps is granted by a set of well-formedness constraints to be satisfied while creating the mockups and writing the steps, as prescribed by DUSM method [RLRC18]. An example of a simple screen mockup associated with a use case step is shown in Figure 3.4; the AddEntryPage mockup is linked to step 3 and contains all the web elements and all the entered data to perform the step interaction.

The web elements of the screen mockups affected by the interactions in use cases steps must be made explicit. This is achieved by annotating the interactions in the glossary with the locators pointing to the specific web elements inside the DOM of the HTML page. Locators are needed in order to retrieve the web elements, once the test scripts are generated from the specification (e.g., to find the link that must be clicked) [LCRT16]. As shown in the meta-model of Figure 3.2, annotations have form [kind: $\text{locator}_1, \ldots, \text{locator}_n$], where kind represents the kind of interaction performed over the web elements (e.g., enter, click), and $\text{locator}_1, \ldots, \text{locator}_n$ are the values needed to identify them. As discussed in Chapter 2, different types of locators exist [LSRT16], based, for instance, on web elements identifiers, names or XPath; in this work, for the sake of simplicity, locators are limited to identifiers. In Figure 3.3, the interaction enters entry details Name and Phone, used in the step 3 of the use case shown in Figure 3.4, is annotated by [enter: "name", "phone"], stating which web elements of AddEntryPage mockup will be used to perform it (i.e., the text fields located by "name" and "phone" identifiers).

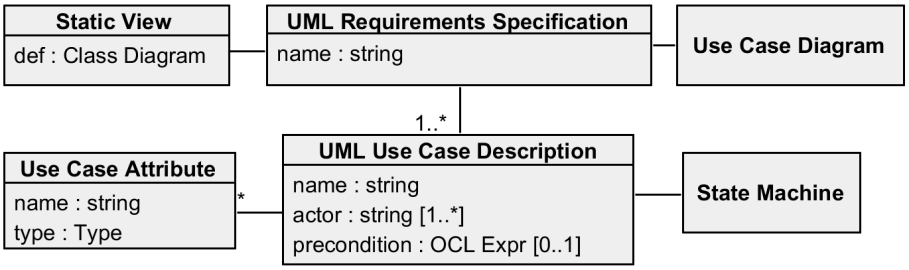
3.4 UML-BASED REQUIREMENTS SPECIFICATION

The UML-based requirements specification has a similar structure of the textual one, even though the parts are expressed using the UML constructs instead of plain text. Then, it consists of a use case diagram, a description of each use case, given by a state machine with associated screen mockups, plus a static view (i.e., a class diagram) defining the used data, the attributes, the related

operations, and the interactions among the actors and the WebApp. The UML-based requirements specification is again precisely defined by a meta-model (see Figure 3.5).

In the following, the use case diagram is omitted, since already introduced in Section 3.3.1.

Figure 3.5: The UML-based requirements specification meta-model.



3.4.1 Static View

The *static view* is a class diagram basically equivalent to the glossary part of the textual requirements specification. It contains: **Datatypes** for the used data, **Web App** class, **Actors** classes, and **Operation** class. A portion of the PhoneBook static view is shown in Figure 3.6.

Datatypes: the UML datatypes defining the data needed to express the requirements. For example, in Figure 3.6 we have Name and Phone, each one containing string values.

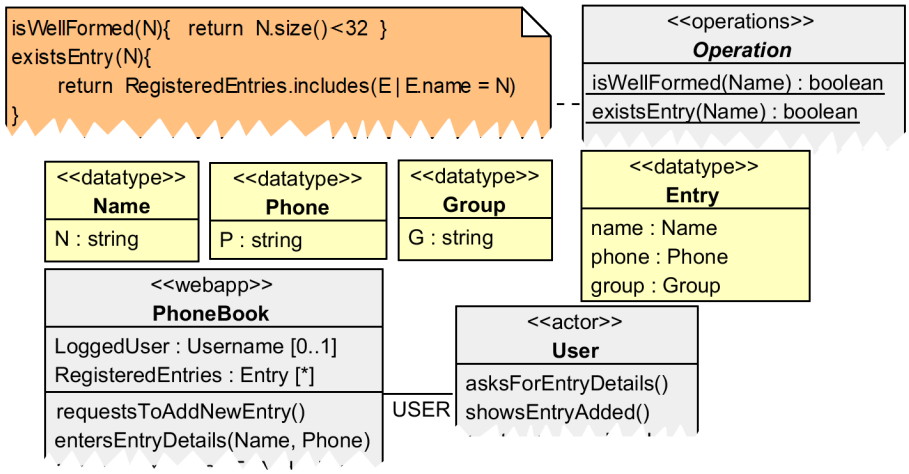
Web App: the UML class modeling the WebApp. It is stereotyped by ≪webapp≫ and contains the attributes describing its updatable state and the interactions performed by the actors towards the WebApp. For example, in Figure 3.6, PhoneBook class has RegisteredEntries attribute, storing all the entries in the WebApp, and entersEntryDetails interaction, called by actors whenever a new entry is added.

Actors: the UML classes modeling the actors, hence containing the interactions performed by the WebApp towards them. Each actor class is stereotyped by ≪actor≫ and must be connected to the WebApp class by an association, named as the actor itself. For example, in Figure 3.6 User class has showsEntryAdded interaction, used by the WebApp to inform the user whenever an entry is added successfully.

Operation: the class containing all the UML operations performed over data and attributes, stereotyped by ≪operations≫. The operations within the class can have or not a return type, and their definitions are given in attached notes and expressed using **Action Language for Foundational UML (ALF)** notation[3]; in the UML terminology, the definitions in the notes are

[3] http://www.omg.org/spec/ALF/1.0.1/PDF/

Figure 3.6: A portion of PhoneBook static view.



the methods associated with such operations. For example, in Figure 3.6 the note attached to isWellFormed operation of the *Operation* class describes whenever a Name N can be considered well-formed, in a similar way to the glossary part shown in Figure 3.3.

3.4.2 Use Cases Descriptions

In the UML perspective, the description of a use case includes some information analogous to those of the textual use cases, but the behavior of the WebApp is here described by a state machine instead of a set of scenarios. An example of a PhoneBook state machine with associated descriptions, corresponding to the use case shown in Figure 3.4, is given in Figure 3.7.

The *use case attributes* are declared in a note stereotyped by ≪attributes≫, and have form "name : type", where the type is a datatype defined in the static view (see Figure 3.6).

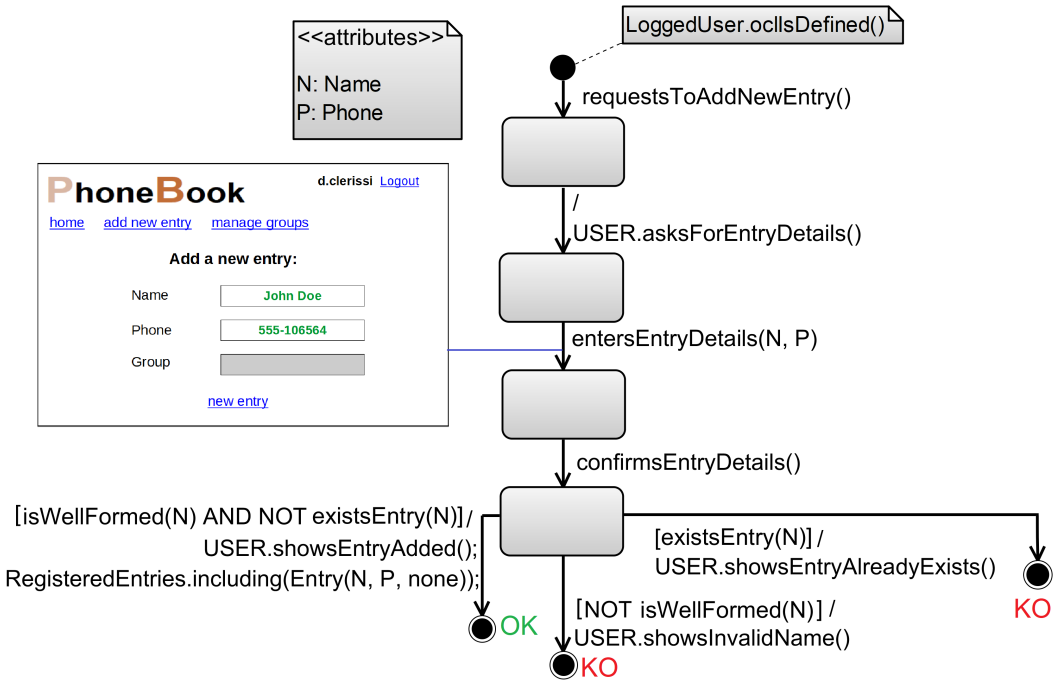
The *preconditions* are **Object Constraint Language (OCL)** expressions[4] put in notes attached to the starting state of the state machine. In Figure 3.7, the precondition refers to oclIsDefined predefined OCL operation and is equivalent to the textual one given in Figure 3.4.

The transitions of the state machine represent the interactions among the actors and the WebApp, and have one of the following forms:

- *Interaction* [*Condition*] / *Effect**, if the transition corresponds to an interactions from an actor towards the WebApp. *Interaction* is an event built by an interaction of the WebApp class, *Condition* is a boolean OCL expression, and *Effect** are either calls of operations of the *Operation* class or basic UML actions, in any case updating the WebApp state. In Figure 3.7, the third

[4] https://www.omg.org/spec/OCL/2.4/PDF

Figure 3.7: PhoneBook Add Entry state machine.



transition is an event built by entersEntryDetails interaction of the WebApp, which uses the declared attributes (i.e., Name N and Phone P) to add a new entry. The transition is semantically equivalent to step 3 of Figure 3.4.

– [*Condition*] / ACTOR.*Interaction* ; *Effect**, if the transition corresponds to an interactions from the WebApp towards an actor. *Interaction* is an event built by an interaction of the Actor class, as determined by the ACTOR association, while the other parts are the same as before. In Figure 3.7, the last transition on the left includes: a condition calling some operations (i.e., isWellFormed and existsEntry) over the entered Name N, the showsEntryAdded interaction of the User class, and the effect of updating the current WebApp state with the entered entry, by calling a basic UML action (i.e., including). The transition is semantically equivalent to step 5 of Figure 3.4.

Similarly to use cases, *postconditions* can optionally be associated with successful paths (i.e., those ending in a state labeled by OK, see Figure 3.7), as OCL expressions put in notes attached to the ending states.

3.4.3 Screen Mockups

In the UML perspective, screen mockups are associated with the transitions and the states of the state machines modeling the use cases behaviors. More specifically, if the transition corresponds

to an interaction of the WebApp towards an actor, a single screen mockup is linked to the transition target state. Instead, if the transition corresponds to an interaction of an actor towards the WebApp, then at most two mockups can be linked: one to the transition source state, and one to the transition arrow head. At least one mockup is needed for each transition. An example of a screen mockup attached to a transition arrow head is given in Figure 3.7.

As well as in the textual perspective, screen mockups must be consistent with the transitions they are linked to [RLRC18]. Moreover, the web elements of the screen mockups that are affected by the interactions in the state machines (e.g., entersEntryDetails of Figure 3.7) must be explicitly referred in the static view, where such interactions are defined. In the UML perspective, this connection is achieved by employing *tagged values*. The *tagged values* encapsulate the kind of interaction performed over the web elements involved in the interaction and strings representing their locators, like the concept of annotations in the textual perspective (see the Annotation class linked to Interaction class in Figure 3.2). The form adopted by tagged values for an interaction is $\{\text{kind} = \text{locator}_1, \ldots, \text{locator}_n\}$. For example, entersEntryDetails interaction of the WebApp class is associated with the tagged value {enter = "name", "phone"}, indicating that the data about name and phone will be entered in the text fields identified by "name" and "phone" strings. Notice that the entered tagged values are not displayed in the static view of Figure 3.6, since this depends on the adopted UML modeling tool (i.e., Visual Paradigm[5], in the case of the example, does not explicitly display them).

3.5 TRANSFORMATIONS BETWEEN TEXTUAL AND UML-BASED REQUIREMENTS SPECIFICATIONS

The transformations are initially described from a high level perspective, and then refined in details by a decomposition stage, where additional sub-transformations are involved.

Each (sub-)transformation shows how a source entity (e.g., a use case step) is transformed into a target entity (e.g., a state machine transition). The procedure of abstractly describing transformations from source to target universes has been inspired by Tiso *et al.* [TRL14].

A (sub-)transformation is characterized by a name, an informal description in natural language declaring its goal, and a graphical representation of how source entities are transformed into target ones, including additional calls to further sub-transformations, in case a decomposition stage is needed.

In the following, only a sketch of the main (sub-)transformations is presented. Also, since **`Text2UML`** and **`UML2Text`** are one the inverse of the other, only **`UML2Text`** is discussed.

[5] `https://www.visual-paradigm.com/`

UML2Text (and inverse **Text2UML**) will be completed and implemented using the **ATLAS Transformation Language (ATL)** of the Eclipse Modeling Project[6], by now the standard for *Model2Model* transformations and also highly supported, well-documented, and integrated in Eclipse IDE.

3.5.1 UML2Text

UML2Text transformation, and the inverse **Text2UML**, shown as the bi-directional grey arrow in Figure 3.1, are *Model2Model* transformations between textual and UML-based requirements specifications.

UML2Text transforms a UML-based requirements specification into a textual one and is composed of several sub-transformations, each one handling a different part of it. The main transformation is defined on top of Figure 3.8: on the left, the source UML-based requirements specification; on the right, the target textual requirements specification, where each part is generated by sub-transformations calls.

More specifically, **UCD** transforms a UML-based use case description (i.e., a UML state machine and additional artifacts, see Figure 3.7) into a textual one, while **Glossary** transforms the datatypes, the attributes, the operations and the interactions defined in the static view into glossary entries. The use case diagram is instead kept unaltered. Again, further sub-transformations compose **UCD** (Figure 3.8, below), generating the actors, the use case attributes, the preconditions, and the scenarios, respectively.

For instance, **Scenarios** takes a state machine in input and generates the main and the alternative scenarios from its paths, each one characterized by a sequence of transitions from the starting to an ending state. Given a state machine and its corresponding use case, the number of individual paths and scenarios is the same, thus use cases and state machines can be considered *isomorphic* in terms of transformations. The states of a state machine having multiple leaving transitions are the extensions points determining the various scenarios in the corresponding use case description; e.g., the last state in the state machine in Figure 3.7 is the extension point for 5, 5a.1, and 5b.1 use case steps in Figure 3.4.

Among the activities here just sketched, **Scenarios** calls **GenerateStep** (Figure 3.9), which transforms a transition into a step; **Exp**, **Inter**, and **Effs** sub-transformations handle with conditions, interactions, and effects, respectively. As shown in Figure 3.1, screen mockups are not affected by the process; however, since they are part of both requirements specifications, transformations will attach them to the corresponding step (in the textual perspective) or transition/state (in the UML perspective).

[6] https://www.eclipse.org/atl/

Figure 3.8: (Above) **UML2Text**: from a UML-based to a textual requirements specification. (Below) **UCD**: from a UML-based to a textual use case description.

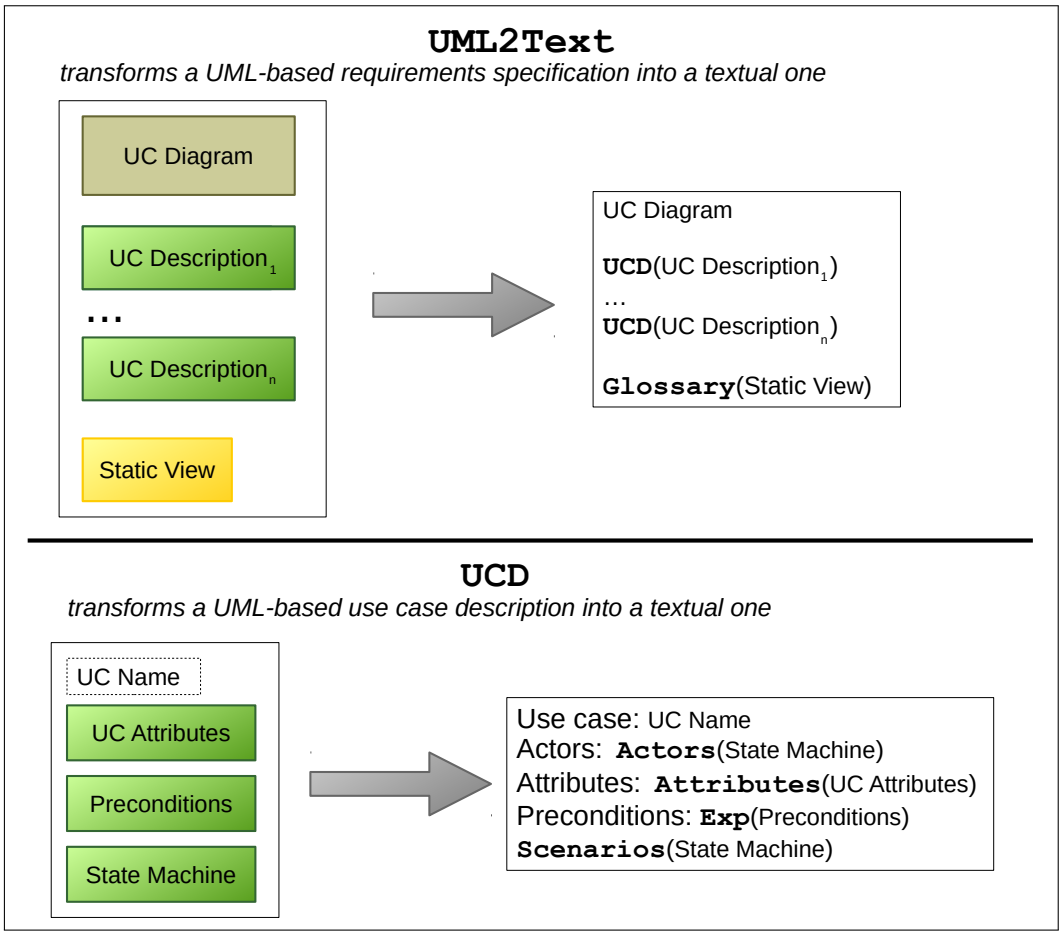
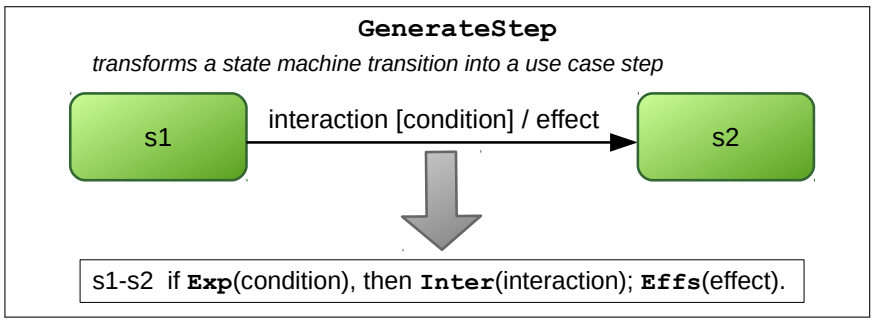


Figure 3.9: **GenerateStep**: from a state machine transition to a use case step (an interaction from an actor towards the WebApp).



3.6 TRANSFORMATIONS FROM UML-BASED REQUIREMENTS SPECIFICATION TO TESTWARE

In the approach, the testware is generated from the UML models, and is characterized by:

- *Test Scripts*: composed of instructions coding the transitions of the state machines paths;
- *Auxiliary Classes*: all the code corresponding to the entities defined in the static view, i.e., data attributes, operations and interactions, used in *Test Scripts* instructions;
- *Test Suite*: the collection of all the *Test Scripts* and the general settings needed for their execution;
- *Screen Mockups*: the HTML pages describing the WebApp GUI over which the *Test Scripts* instructions are performed.

The testware components will be coded in Java, since the language fits well with the state-of-the-practice Selenium WebDriver testing framework[7] [LCRT16] and is supported by Eclipse development environment, which also hosts several *Model2Text* transformations languages and code generators.

`UML2Test` transformation, in the following briefly discussed, will rely on **Acceleo**[8], which is an OMG standard for *Model2Text* transformations and, again, is well-documented and embodied in Eclipse IDE.

3.6.1 UML2Test

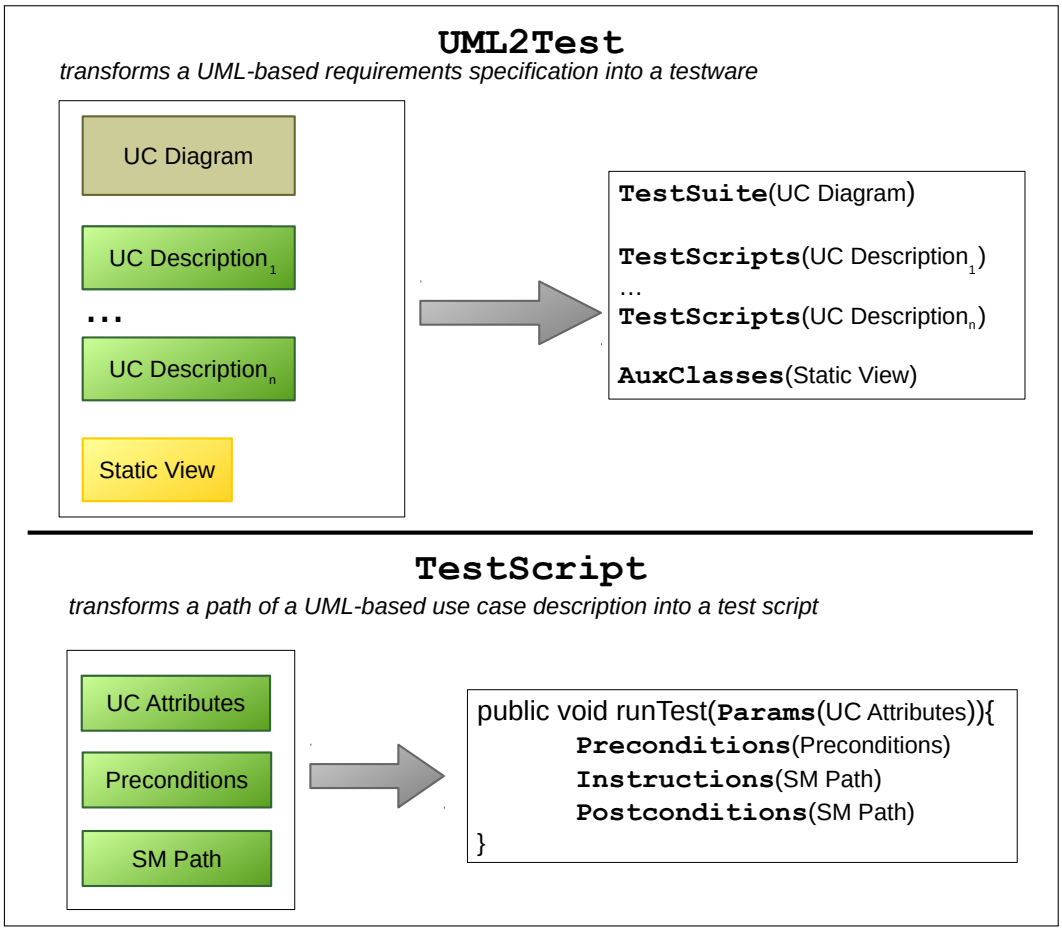
`UML2Test` is the *Model2Text* transformation responsible for the testware generation from a UML-based requirements specification (last grey arrow in Figure 3.1), and is based again on several sub-transformations. The various UML constructs are separately transformed into Java code, as shown on top of Figure 3.10.

More specifically, **`TestSuite`** gives the structure of the test suite, grouping the test scripts together as driven by the use case diagram. **`TestScripts`** transforms a UML-based use case description (i.e., a state machine and additional artifacts) into several test scripts, each one covering a specific path. Finally, **`AuxClasses`** generates the code representing the content of the static view, needed to formulate the test scripts instructions.

`TestScripts` handles the various paths of the state machine representing the behaviors of a use case; different algorithms solving minimum-cost flow problems may be used to extract paths from the state machine (e.g., some of those proposed by Kleinberg and Tardos [KT06]). Thus, it creates a class for all the test scripts separately covering the state machine paths and

[7] `https://www.selenium.dev/documentation/en/webdriver/`
[8] `https://www.eclipse.org/acceleo/`

Figure 3.10: (Above) **UML2Test**: from a UML-based requirements specification to a testware. (Below) **TestScript**: from a path of a UML-based use case description to a Java test script.



calls, for each path, **TestScript** (Figure 3.10, below). **TestScript** transforms a path into a test method of the current class: the use case attributes become the parameters of the method (**Params** sub-transformation), since they represent the entered data, the pre/post conditions are treated as assertions (**Preconditions** and **Postconditions** sub-transformations), and the transitions composing the path are translated into test instructions by **Instructions**. Notice that, since postconditions are specific for successful paths, they are taken directly from the current path. For each transition, **Instructions** differentiates the ones having an actor as subject from those having the WebApp; the latter are treated as assertions, since the WebApp may have to notify/show to the user the details about the content of a web page. Finally, to make test scripts directly executable, a last instrumentation step for feeding the code with proper input data, based on the declared use case attributes, is necessary.

Listing 3.1 shows a simplified Selenium WebDriver test script (i.e., a method) that would be generated from a path of the state machine shown in Figure 3.7. It represents the scenario of adding a valid entry to PhoneBook. All instructions are calls to attributes or methods of the auxiliary classes, generated by **`AuxClasses`** (i.e., PhoneBook, User, and *Operation*), representing the operations over the data and the interactions between the user and the WebApp. Such interactions encapsulate Selenium WebDriver APIs; for example, entersEntryDetails (transition 3) is a method of PhoneBook class and represents the action of entering Name N and Phone P in the corresponding text fields.

Listing 3.1: PhoneBook Add Entry test script (main success scenario).

```
1  public void runTest(Name N, Phone P){
2    assertTrue(PhoneBook.LoggedUser != null); //precondition
3    PhoneBook.requestsToAddNewEntry();        //transition 1
4    assertTrue(User.asksForEntryDetails());   //transition 2
5    PhoneBook.entersEntryDetails(N, P);       //transition 3
6    PhoneBook.confirmsEntryDetails();         //transition 4
7    assertTrue(Operation.isWellFormed(N) && !Operation.existsEntry(N));//transition 5
8    assertTrue(User.showsEntryAdded());       //transition 5
9    PhoneBook.RegisteredEntries.add(new Entry(N, P, null));//transition 5
10 }
```

Chapter 4

Related Work

Many works investigate in the relationships between use cases (and, more generally, requirements) and testing artifacts, and how to get the latter from the former.

Hellmann *et al.* [HHM11] present a Test Driven Development approach for GUI-based applications, where low fidelity prototypes are sketched and linked together through event handlers to activate navigation functionalities. In a second time, interactions on prototypes are recorded to produce test scripts able to drive the development. While the core idea is similar to the one proposed in Chapter 2, some differences exist: the context is not the web, test scripts are not strictly aligned to use cases, and the proposal does not investigate towards improvements in robustness and effectiveness of the generated test suite.

Besson *et al.* [BBC10] propose an ATDD approach for web applications based on user stories. The functionalities of the target web application are mapped into a graph, where each path represents a testing scenario as a navigational sequence of events through pages. Testing scenarios are then validated by the customer and subsequently transformed into executable test scripts. Conversely to Besson *et al.*, the approach of Chapter 2 does not require the graph structure, which is substituted by a simpler recording phase of interactions on screen mockups driven by use cases. Therefore, test scripts are easier to get and to maintain during a web application evolution.

Mugridge [Mug08] developed an extension to the Fit framework[1] to improve expressiveness of story tests (i.e., fixtures workflows based on user stories), automatically coding them into executable test scripts. Similarly to the ATDD approach presented in Chapter 2, user stories can guide test scripts definition and execution. However, the paper does not focus on the web domain and testing evolution. Test scripts need fixtures tables and an actual system to run, while in the approach proposed in Chapter 2 they can be recorded and executed directly on screen mockups.

[1] http://fit.c2.com/

Cucumber[2] is a software used to describe requirements of applications in a structured way and use them as a guidance for the development and testing phases, based on Behavior-Driven Development [SW11]. In Cucumber, the functionalities follow restrained scenarios adhering to the *given, when, then* template, from which test cases instructions are generated. In the approach presented in Chapter 2, use cases are structured enough to make the interactions occurring on the GUI explicit, but are not restrained to a very strict template; hence, the test scripts can be easily generated from the use cases and directly executed on the screen mockups.

Olek *et al.* [OAN14] introduce a Test Description Language to record manual interactions occurring on web GUI sketches, attached to use cases steps, and code them into test cases instructions. Similarly to the approach presented in Chapter 2, use cases are used as a starting point for test scripts definition. However, in this work, the aid of a specific tool to capture the interactions and of a language to represent such interactions are essential.

In the context of Model-Driven Web Engineering (MDWE), Rivero *et al.* [RGR+14] propose an iterative agile-MDWE process based on mockups to support web applications development. HTML mockups are generated from user stories, tagged to explicit widgets semantic, and turned into MDWE models to generate code. The main difference with respect to the approach presented in Chapter 2 is that the work of Rivero *et al.* is not based on the ATDD paradigm. In their paper, user stories are the basis to design screen mockups for next development, although they do not support the testing phase.

The following proposals transform restrained use cases into a state model to generate test cases from it, trying to keep the artifacts synchronized. Yue *et al.* [YAB11] present an automated approach to generate state machines from restrained use cases, according to a set of transformation rules. Then, by means of a model-based testing technique applied over the state machines representing the system, abstract test cases are extracted. Somé implemented the UCEd tool[3] to transform simplified use cases into a state model, from which abstract test cases representing use cases scenarios are extracted. Finally, Jiang *et al.* [JD11] proposed an approach to automatically generate test cases from use cases, whose descriptions are constrained to predefined sentences. Use cases lead to the generation of Extended Finite State Machines (EFSM), where paths corresponding to test cases can be drawn. Changes in the use cases are reflected to EFSMs, hence providing the alignment between requirements and tests.

The approach presented in Chapter 3 differentiates from the aforementioned works in several ways. The final output are executable test cases (i.e., Java code based on Selenium WebDriver testing framework) directly runnable over a web application. The requirements specifications, both textual and UML-based, are made precise and are integrated with the screen mockups, which are essential ingredients empowering the overall comprehension and helping in the subsequent testing process. The glossary of usable terms to formulate use cases sentences is structured but not much restrained, thus provides more freedom and customization.

[2] https://cucumber.io/

[3] http://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCEd.html

Chapter 5

Conclusion and Future Work

The proposals presented in Chapters 2 and 3 try to address some of the challenges pertaining the development and testing activities in the web domain, where requirements rapidly evolve and a suite of aligned effective test cases must be generated soon afterward.

Although a large number of tools and solutions for the web emerged during the years, the majority neglect a fully alignment between requirements specifications and executable test artifacts. Even acceptance test driven development approaches, aimed at generating running portions of web applications from test scripts, may turn difficult to apply, due to the intricateness of existing tools and the lack, at early stage, of a working prototype of the web application to use as a baseline.

Precision in requirements specifications, in the form of structured use cases with interactive screen mockups to clarify the GUI functionalities, may be the basis to early obtain executable test scripts that are fully aligned with a system expected behavior, and reduce the effort to keep the artifacts synchronized.

Chapter 2 introduced a novel approach for developing web applications adopting the ATDD practice. The novelty regards the usage of screen mockups and capture-replay testing tools for easily generating acceptance test scripts, guided by requirements specifications, and able to drive the development of the target web application, before an actual prototype is available. To show its feasibility, the approach has been successfully applied to re-develop and test an existing web application. The adoption of the approach resulted in some lesson learned. It is applicable to the development of medium-size web applications with a limited effort, and the process is generalizable also for other web applications of similar complexity. The requirements expressed by means of use cases and screen mockups are helpful to guide test suite development, since their scenarios naturally describe the user interactions more than concise user stories may do; also, it is not too problematic to pay a little more effort in screen mockups creation, given that they can be reused as the basis for the GUI development. The structure of use cases can support test artifacts creations in several ways: use cases extensions suggest what must be tested and which

scenarios should be considered, while pre and post conditions allow to determine the starting point from which to record a test script and the conditions/properties that must be checked. Moreover, the early addition in screen mockups of meaningful web elements locators guarantees to have useful and robust test scripts during software evolution and maintenance. More comprehensive studies (comparative experiments, case studies, smart input data selection, and evaluation of the actual industrial applicability) should be conducted to gather feedback on the effectiveness and usefulness of the approach.

Chapter 3 sketched an approach for web applications aimed at defining a precise requirements specification, either textual or expressed using the UML, to generate from it a functionally complete set of test scripts, exploiting existing *Model2Model* and *Model2Text* transformation tools. Textual and UML-based requirements specifications are semantically equivalent, thus the analyst is left free to choose from which one to start. Two transformations are employed to automatically move between the textual and the UML perspectives, and an additional transformation generates the testware from UML. The approach is currently tailored for web applications, whose requirements should be precisely specified (e.g., banking, e-commerce/payments systems, government services), having the functionalities clearly described in terms of GUI and interactions - hence using the requirements specification as a sort of user manual - and with the need for an intensive testing process to enforce their reliability. Future work will include the implementation of the sketched transformations and their integration in a prototype tool, aimed at reducing the manual effort needed to apply the approach as much as possible. Making requirements specifications precise, even if tool-assisted, is indeed an onerous task, but the preliminary effort is rewarded in the last stage of the approach, where the testing artifacts are automatically derived and kept aligned with the requirements. The cost of applying the approach will be empirically compared against different approaches based on manual or semi-automatic test generation. Moreover, the maintainability cost of the requirements specifications and of the testware during the web application natural evolution will also be investigated.

Part III

Quality Assurance Approaches for the IoT Domain

Chapter 6

A Set of Empirically Validated Development Guidelines for Improving Node-RED Flows Comprehension

Internet of Things (IoT) systems are rapidly gaining importance in the human society, providing a variety of services to improve the quality of our lives, involving complex and safety-critical tasks; therefore, assuring their quality is of paramount importance. Node-RED[1] is a Web-based visual tool inspired by the flow-based programming paradigm [Mor10], built on Node.js[2], and recently emerged to support the development of IoT systems in a simple manner.

The community behind Node-RED is quite active and artifacts sharing is strongly encouraged. Thus, the Node-RED flows developed and submitted to public usages are expected to be easy to comprehend and integrate within already existing systems, also in preparation of future maintenance and testing activities. Unfortunately, no consolidated approaches or guidelines to develop comprehensible Node-RED flows currently exist.

This chapter presents a set of proposed guidelines to help the Node-RED developers in producing flows that are easy to comprehend and use. An experiment has been designed and conducted to evaluate the effect of the guidelines in Node-RED flows comprehension. Results show that the adoption of the guidelines significantly reduces the number of errors (p-value = 0.00903) and the time required to comprehend Node-RED flows (p-value = 0.04883).

The content of this chapter has been accepted for publication in the *15th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE 2020) [CLR20].

[1] https://nodered.org/
[2] https://nodejs.org/

6.1 INTRODUCTION

Recently, Node-RED has emerged as a practical solution to easily develop and share IoT systems.

Node-RED is a Web-based tool inspired by the flow-based programming paradigm [Mor10] and built on top of the Node.js framework. In Node-RED, a *node* represents a black-box component, implementing part of a device logics or, more generally, of a service provided by a system. Nodes are largely configurable, and are wired together into *flows*, once they have to cooperate/communicate, by exchanging data *messages*. Since the core language is Javascript, in Node-RED every entity that brings information, from messages to nodes to even flows, is represented as a JSON object (i.e., a sequence of key/value pairs surrounded by curly braces[3]).

Daily, nodes and flows are developed and uploaded to the Node-RED library by the developers participating in the community (over 2000 nodes in 2019[4]), as solutions to general or specific problems, and anyone may download part of this content to integrate it within existing systems. Nodes can execute a variety of tasks, like reading values from a database, running a JavaScript function, receiving the feeds from a Twitter account, establishing a communication between two devices using the MQTT protocol, and more.

As the flow-based programming paradigm prescribes [Mor10], the nodes are black-box components that hide all the implementation details to the final user (i.e., basically, JavaScript functions and graphical features). The developer can select the nodes she desires and wire them together in order to build the system she wants, without having the complete knowledge of each node setting.

Like any other programming tool and language, Node-RED lets the developer to choose her own *programming style* while implementing new nodes and flows. Since Node-RED is a visual tool, along with the programming style, there is also the **comprehensibility factor** related to the *graphical style* adopted for wiring the nodes together to compose the Node-RED flows, as well as for carefully choosing meaningful names for the nodes; in general, this is a problem more frequently found at design stage. The lack of a disciplined approach as a guidance for developing Node-RED flows could result in messy "spaghetti" artifacts, very hard to comprehend and use, which may produce unexpected outcomes when they are integrated into further complicated systems, without mentioning the pain of maintaining and testing them.

Up to now, no consolidated approaches supporting Node-RED developers in producing reusable and comprehensible flows exist, and only few basic and unofficial attempts proposing best practices and design patterns are available[5].

This chapter outlines a preliminary set of guidelines formulated in order to produce Node-RED flows that are easy to comprehend by construction, and more suitable to reuse, maintain and

[3] https://www.w3schools.com/js/js_json_objects.asp

[4] https://flows.nodered.org/

[5] https://medium.com/node-red/node-red-design-patterns-893331422f42

test. The benefits of adopting the proposed guidelines have been investigated by means of an experiment involving ten master students, where two selected Node-RED systems, each one developed with and without the guidelines, are compared in a comprehension scenario.

In Section 6.2 of this chapter the guidelines are described, while some Node-RED comprehensibility issues pertaining the two selected Node-RED systems are introduced in Section 6.3, showing how the guidelines can be applied on them. Finally, the experiment and the results are discussed in Sections 6.4 and 6.5, respectively.

6.2 THE GUIDELINES

The proposed guidelines address some common Node-RED comprehensibility issues which may emerge while developing flows or trying to understand and integrate flows provided by an external source (e.g., the Node-RED community library). Issues may concern confusing nodes names, hidden loops and loss of messages in flows, lack of conditional statements, unexpected inactive nodes, and more. More details about issues are provided in Section 6.3.3. The guidelines aim at supporting Node-RED developers in producing flows that are easy to comprehend by construction, and suitable for future reuse, maintenance and testing activities.

The guidelines have been inspired by several design works addressing systems quality using UML and Business Process Model and Notation (BPMN) [Amb05, MRvdA10, Unh05, RLR11, RLRA12], and by the personal experience acquired in IoT systems design and Node-RED flows development [CLRR18, LCO+18]. UML is one of the most used notational languages [RLR14], and differs to Node-RED in many aspects: while UML works at design level and describes the static and dynamic details of a system, Node-RED is an executable visual language used to implement, execute and deploy a working system. The constructs they use are quite different, as well as their syntax and semantics. Nevertheless, as experienced, in practice some design and technology-independent principles can be inherited from UML even to solve specific Node-RED issues [CLRR18, LCO+18].

To better comprehend the Node-RED terminology and the issues that the guidelines try to address, Table 6.1 recaps a short list of terms and definitions, extracted and elaborated from the Node-RED official documentation[6].

The proposed guidelines can be classified into four types, based on the comprehensibility issues they address: **Naming**, **Missing Data**, **Content**, and **Layout**.

[6] https://nodered.org/docs/

Table 6.1: Node-RED Essential Terms and Definitions.

Term	Definition
Node	The basic Node RED component, representing (part of) the logics of a service or a functionality. Each node has a type describing its general behavior and a set of custom properties.
Flow	The logical way the nodes are wired, expressing how they collaborate by exchanging messages.
Sub-Flow	A self-contained logical portion of a flow, contributing to its completion.
Wire	The edge used to graphically connect two nodes of a flow.
Pin	The input/output port of a node, where a wire enters/leaves.
Message	A data object exchanged by some nodes, characterized by a sequence of configurable properties.
Global/Flow Variable	A variable defined in a node and visible by all the flows (if global variable) or by just the one containing that node (if flow variable).

6.2.1 Naming

Node Name Behavior (NNB) Each Node-RED node should have a unique (unless a duplicate of another existing node) and meaningful name, suggesting its high-level behavior [LDCD06]. The name of a node should make explicit the *action(s)* performed by the node and the *object(s)* receiving such *action(s)*. An *object* may refer to a message property or a global/flow variable, written in upper-case to be more visible within the flow [RLRA12].

Flow Name Behavior (FNB) Each Node-RED flow should have a unique and meaningful name, summarizing in a very concise way its high level behavior [LDCD06].

6.2.2 Missing Data

Node Effective Contribution (NEC) By adapting to Node-RED some of the terms used by Ambler [Amb05], there should neither exist *black hole* nodes nor *miracle* nodes. A black hole node is a node with no leaving wires but output pins > 0, which means that the node output might be lost or unused by the flow, while a miracle node is a node with no entering wires but input pins > 0, which means that the node cannot be explicitly activated or is missing some expected data.

Conditions Consistency and Completeness (CCC) The conditions of every switch node (i.e., a core Node-RED node basically implementing the switch/if constructs of every programming language, and used to route the messages by evaluating a set of conditional statements over glob-

al/flow variables or message properties[7]) should not overlap and be complete (i.e., their disjunction returns true) [Amb05, RLRA12], in order to handle separately all the possible scenarios.

6.2.3 Content

Sub-Flows Relatedness (SFR) The sub-flows composing a flow should be logically related among each others [LDCD06], following the design principle of high cohesion and low coupling [Mar03]. Two sub-flows F1 and F2 are logically related if, e.g.,:

- F1 and F2 describe (part of) the behavior of the same device;
- F1 and F2 contribute to the same service/functionality;
- F1 and F2 share some variables or other data;
- F1 is activated by F2 or vice versa;

Flow Content (FC) If a flow is overpopulated, its content should be simplified [MRvdA10, Unh05, Amb05], by identifying its sub-flows, and either: (a) physically split and connect them together through link nodes (i.e., a core Node-RED node used to add a virtual wire between two sub-flows[8]), or, (b) collapse them into corresponding sub-flow nodes (i.e., a core Node-RED node used to collect sub-flows to favor reuse and reduce layout complexity[9]). Since Node-RED flows design and development phases are strongly related due to the visual nature of the tool, as in the case of more general design activities, there is a positive correlation between flows size and complexity. Following this line of reasoning, a flow is then classified as overpopulated if the number of nodes it contains is equal or above 50 [MRvdA10].

6.2.4 Layout

Wiring Style Consistency (WSC) The wires connecting the nodes should follow a consistent wiring style, to differentiate main/correct scenarios from exceptional/wrong ones [RLR11, RLRA12, Amb05, Unh05]. Since Node-RED flows may handle several scenarios, as it happens for classic programming languages concerning conditional statements, different wiring styles may be adopted within the same flow. For example, a "straight, from left to right, top-down" style to wire all the nodes participating in a correct scenario, and a "cascade" style to wire all the nodes participating in a wrong scenario.

[7] https://nodered.org/docs/user-guide/nodes#switch
[8] https://nodered.org/blog/2016/06/14/version-0-14-released
[9] https://nodered.org/docs/user-guide/editor/workspace/subflows

Wiring Style Tidiness (WST) The wires connecting the nodes should be long enough to clearly show the starting/ending nodes and avoid any overlapping, whether possible [Amb05]. The node joining multiple wires should be placed at the level of the node where such wires originated.

6.3 THE SELECTED NODE-RED SYSTEMS

To conduct the experiment, later discussed in Section 6.4, two existing Node-RED systems were selected; the systems were developed by former students of the master course Data Science and Engineering held in Genova, Italy, as part of a last year project.

The systems, named **DiaMH** and **WikiDataQuerying**, present some common Node-RED comprehensibility issues derived from an undisciplined usage of the tool (i.e., the teacher of that course was not involved in this research and thus students developed the systems without following any guideline).

6.3.1 DiaMH System

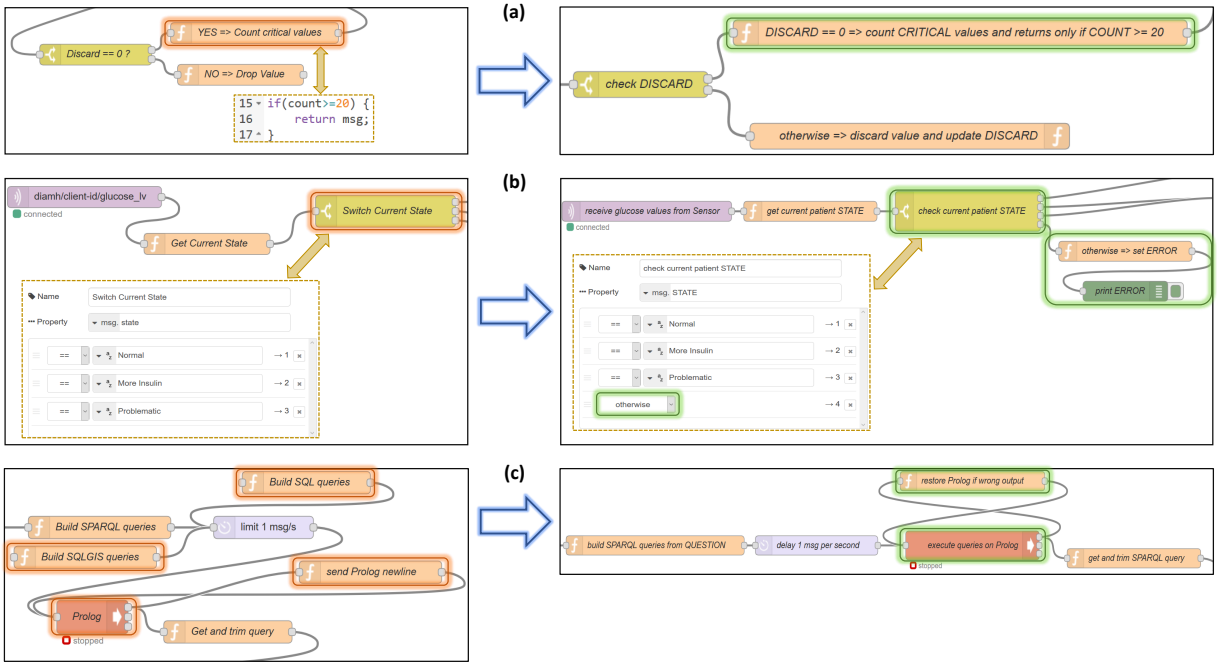
DiaMH is a simulated Diabetes Mobile Health IoT system which monitors a diabetic patient by collecting glucose values using a wearable sensor, sends notifications to the patient's smartphone about the monitored data, and, based on some logical computations involving a cloud-based healthcare system and realistic data patterns, determines the patient's health state (i.e., Normal, More Insulin required, or Problematic) and, when needed, orders insulin injections to a wearable insulin pump. It consists of 71 nodes and 63 wires.

6.3.2 WikiDataQuerying System

WikiDataQuerying is a web query service used to select textual geo-spatial questions from a predefined list shown in a HTML page, and query WikiData[10] knowledge base, by first restructuring the selected questions into SPARQL query language and formatting them using a Prolog grammar. The results of the queries can be triples adhering to the Resource Description Framework (RDF) language or boolean answers. It consists of 25 nodes and 27 wires.

[10] https://www.wikidata.org

Figure 6.1: Issues in Node-RED Systems (left) and Guidelines Application (right).



6.3.3 Applying Guidelines to Node-RED Systems

For the selected Node-RED systems, only some of the aforementioned guidelines have to be applied. However, even such simple systems can hide several comprehensibility issues. Despite their simplicity, involving mainly core Node-RED nodes, DiaMH works in the thorny context of the healthcare and WikiDataQuerying must provide prompted feedback to the user's requests. Therefore, producing Node-RED flows that adhere to the proposed guidelines may improve the comprehensibility level during flows inspection and integration, and facilitate the subsequent engineering stages, such as maintainability and testing.

Just a few of the issues found from a high-level flows analysis of the systems are shown in Figure 6.1 and discussed in the following.

Most of the nodes names do not clarify their behaviors, forcing the developer to inspect the nodes contents in order to comprehend them. Therefore, **NNB** can be applied to clarify the nodes purposes, by making explicit in their names the performed actions and the used variables in uppercase (see nodes names in Figure 6.1, changed from left to right).

Sub-flow **(a, left)** of Figure 6.1 presents the function node (i.e., a core Node-RED node used to implement customized JavaScript functions[11]) named Yes => Count critical values, which performs several actions and then sends a message to subsequent nodes (not shown in Figure 6.1) only when a certain condition holds (i.e., if count $\geq$ 20, see lines 15-17 in Figure 6.1 left associated with the node); from an unaware Node-RED developer perspective, this may unexpectedly block the execution of the sub-flow until the condition is satisfied, even if the nodes are graphically connected by means of wires. This comprehensibility issue emerged in several topics posted on the main Node-RED forum[12] and can be solved by applying **NNB**, as previously mentioned, by renaming the function node as, e.g., DISCARD == 0 => count CRITICAL values and returns only if COUNT >= 20, in order to make explicit: the condition it satisfies from the preceding switch node (i.e., DISCARD == 0), the core behavior (i.e., count CRITICAL values), and the hidden condition for the message to return (i.e., returns only if COUNT >= 20). Sub-flow **(a, right)** of Figure 6.1 is the result.

In sub-flow **(b, left)** of Figure 6.1, the switch node Switch Current State hides a severe issue: it considers only three possible values for the msg.state variable, but the check will fail and idle the sub-flow execution if any unexpected event sets the variable to a different value before the switch node. This problem was solved in the dawn era of Node-RED[13] by introducing an "otherwise" entry to handle all the alternative conditions, but the average Node-RED developer may still miss to use it in favor of a more explicit, but incomplete set of conditions[14]. By applying **CCC**, the "otherwise" condition is added to the switch node (see the change in the configuration panel from left to right in Figure 6.1), while the application of **WSC** displays in a cascaded wiring style the newly introduced exceptional scenario. Sub-flow **(b, right)** of Figure 6.1 is the result.

Sub-flow **(c, left)** of Figure 6.1 presents several issues. First, the wires do not follow any consistent wiring style (e.g., building the flow by wiring nodes from top to bottom or from left to right, avoiding crossing wires), which reduces the overall comprehensibility[15]. Second, a loop is generated between Prolog and send Prolog newline nodes; although in this scenario finding the loop is rather simple, it may be harder to detect and produce a weird outcome or overheat the CPU, when more wires are involved[16]. Third, the sub-flow shows three function nodes connected through output pins to limit 1 msg/s node, but only Build SPARQL queries actually contributes to the flow, since the other two nodes have no entering wires, making them inactive; this case in particular is easy to detect, but could be hard to understand for a novel Node-RED developer, in case she forgets to trace a wire between two nodes to specify the input source or the output destination of a node, without receiving any explicit warning from the Node-RED environment. This last issue often arises when there is a need for debugging a flow and some nodes have to

[11] https://nodered.org/docs/user-guide/nodes#function
[12] https://discourse.nodered.org/t/function-node-stopping/7017
[13] https://github.com/node-red/node-red/issues/88
[14] https://discourse.nodered.org/t/switch-node-not-consistent/11908
[15] https://discourse.nodered.org/t/help-simplifying-flow/8765
[16] https://discourse.nodered.org/t/cpu-hogging-to-100/2944

be temporary disconnected from it[17]. By applying **WSC** and **WST**, a more consistent and tidier wiring style is generated, to highlight the loop and avoid further entangles, while **NEC** is used to remove the miracle nodes originally named Build SQL queries and Build SQLGIS queries (i.e., those having input pins but no entering wires, in fact inactive). Sub-flow **(c, right)** of Figure 6.1 is the result.

6.4 EXPERIMENTAL EVALUATION

Based on the Goal Question Metric (GQM) template [VSBCR02], the main goal of the experiment can be defined as follows: "Evaluate the effect of the guidelines in Node-RED flows comprehension", with the purpose of understanding if they are able to improve the *comprehension level* of Node-RED flows and the *time* required to complete tasks pertaining such flows; therefore, consequently, the overall *efficiency* is computed as: *comprehension level* ÷ *time*.

The *perspective* is of: (a) Node-RED developers, using it for their own purpose and/or sharing artifacts with the community, who may be interested to consider a disciplined technique to develop Node-RED flows using the guidelines, (b) teachers and instructors interested to offer courses and tutorials on Node-RED, and, (c) researchers interested in focusing their research activities and study improvements or constraints to the Node-RED language.

Thus, the research questions are:

RQ1. Does the comprehension level of Node-RED flows vary when the guidelines are applied?

RQ2. Does the comprehension time of Node-RED flows vary when the guidelines are applied?

RQ3. Does the efficiency of completing tasks pertaining Node-RED flows vary when the guidelines are applied?

To quantitatively investigate the research questions, an ad-hoc questionnaires containing 16 comprehension questions for each experimental objects was employed. The comprehension level of Node-RED flows was measured as the number of correct answers on the total, the comprehension time was measured as the time required to provide such answers, and the efficiency as the ratio between the comprehension level and the comprehension time (i.e., the number of correct answers divided by the time is a proxy for measuring the efficiency construct).

Table 6.2 summarizes the main elements of the experiment, following the guidelines by Wohlin *et al.* [WRH+12].

[17] https://discourse.nodered.org/t/how-to-comment-out-a-node/1106

Table 6.2: Overview of the Experiment.

Goal	Evaluate the effect of the guidelines in Node-RED flows comprehension
Quality focus	Pertaining Node-RED tasks, it is evaluated: (i) Comprehension (ii) Time (iii) Efficiency
Context	**Objects**: DiaMH and WikiDataQuerying Node-RED systems **Participants:** 10 Computer Science master students
Null Hypotheses	(i) No effect on comprehension (ii) No effect on time (iii) No effect on efficiency
Treatments	Non-compliant to guidelines ($^{-}$) and Compliant to guidelines ($^{+}$) Node-RED flows
Dependent variables	(i) TotalComprehension to complete Node-RED tasks (ii) TotalTime to complete Node-RED tasks (iii) TotalEfficiency to complete Node-RED tasks

In the following, treatments, objects, participants, experiment design, hypotheses, variables, procedure, and other aspects of the experiment, are discussed.

6.4.1 Treatments

The experiment has one independent variable (main factor) and two treatments: *Non-compliant* and *Compliant* Node-RED flows. Non-compliant Node-RED flows (in the following, characterized by symbol $^{-}$) are those produced without following the guidelines, while compliant Node-RED flows (in the following, characterized by symbol $^{+}$) are those produced following the guidelines.

6.4.2 Objects

The objects of the study are DiaMH and WikiDataQuerying systems, presented in Section 6.3. Both were developed by former students of another course. Each object was limited to just a comparable (in size and complexity) flow of the original behavior, consisting of 20 nodes and 23 wires for DiaMH and 21 nodes and 21 wires for WikiDataQuerying, employing mostly Node-RED core nodes.

The flows of the systems were carefully inspected and tested; notice that these two flows correspond to the non-compliant treatment ($^{-}$), since the guidelines were not adopted during their implementations. Then, the guidelines discussed in Section 6.2 were applied, producing two equivalent

Table 6.3: Experimental Design ($^+$ for Compliant treatment, $^-$ for Non-Compliant treatment).

	Group A	**Group B**	**Group C**	**Group D**
Task 1	DiaMH$^+$	DiaMH$^-$	WikiDataQuerying$^+$	WikiDataQuerying$^-$
Task 2	WikiDataQuerying$^-$	WikiDataQuerying$^+$	DiaMH$^-$	DiaMH$^+$

compliant versions ($^+$), again inspected and tested. In total, four Node-RED flows for executing the experiment were obtained: DiaMH$^-$, DiaMH$^+$, WikiDataQuerying$^-$, WikiDataQuerying$^+$.

6.4.3 Participants

Ten Computer Science master students of the University of Genova (Italy) were involved. They were attending a course on advanced software engineering; the total number of students enrolled in the course was 15, which is basically the average number of students enrolled in any Computer Science Master Course held in Genova.

The participants had average knowledge of Software Engineering, UML and JavaScript (the Node-RED core programming language), and few experience in Node-RED and flow-based programming, that was provided in another course related to Node-RED development.

6.4.4 Experiment Design

Before the experiment, all the participants were involved in a 4-hours lecture split in two days about Node-RED theory and practice using the tool. Participants were provided with material to understand the main Node-RED core nodes, samples of flows and sub-flows to reproduce/change, and questions to answers about comprehensibility issues of the flows similar to those that were asked for the later experiment. Participants were not informed about the guidelines, and therefore, about the treatments.

Due to the limited number of participants (only ten), a counterbalanced experiment design was adopted for ensuring each participant to work in two tasks on the two different objects, receiving each time a different treatment. Since participants had the same experience in Node-RED, acquired by attending another course, they were randomly split into four groups (see Table 6.3), balancing the representatives for each group. Each participant had to work first on **Task 1** on an object with a treatment, then in **Task 2** on the other object with the other treatment.

6.4.5 Dependent Variables and Hypotheses Formulation

The experiment had three dependent variables, on which the treatments were compared measuring three different constructs to answer the three research questions: (a) *Comprehension* of the Node-RED flows (measured by variable *TotalComprehension*), (b) *Time* required to answer the questions pertaining the Node-RED flows (measured by variable *TotalTime*), (c) *Efficiency* in completing the tasks pertaining the Node-RED flows (measured by variable *TotalEfficiency*).

For each treatment:

- *TotalComprehension* was computed by summing up the number of correct answers of each participants;
- *TotalTime* was computed as the difference between the stop time of the last question and the start time of the first question, where timing was tracked down in the time sheet by each participant;
- *TotalEfficiency* was derived by the two previously computed variables, as:

$$TotalEfficiency = \frac{TotalComprehension}{TotalTime}$$

Since no previous empirical evidence pointing out a clear advantage of one treatment versus the other could be found, the following three null hypotheses were formulated as non-directional, with the objective to reject them in favor of alternative ones:

- H_{0a}: $TotalComprehension^{-} = TotalComprehension^{+}$
- H_{0b}: $TotalTime^{-} = TotalTime^{+}$
- H_{0c}: $TotalEfficiency^{-} = TotalEfficiency^{+}$

6.4.6 Material, Procedure and Execution

To estimate the comprehensibility of the tasks to provide to the participants and the time required to complete them, a pilot experiment with three master students in Computer Science not involved in the experiment was conducted. On average, the time required to complete both tasks was about 105 minutes, with 5 errors. Given such results, the tasks were reworked to try to remove any ambiguity.

Then, the material was uploaded on the Moodle module of the course from which the participants were selected, consisting, for each group of Table 6.3, of: two Node-RED flows (one per system/treatment), two questionnaires containing 16 questions each, and a post-questionnaire to fill after the completion of the two questionnaires containing 7 further questions.

Each questionnaire presented exactly 7 open questions and 9 multiple choices questions, in order to keep the perceived complexity of both tasks as equivalent as possible. Questions ranged from comprehending the general behavior and the structure of the provided Node-RED flows, like identifying the names and the number of nodes involved in certain activities, detecting the presence of loops and missing conditions in switch nodes, counting the number of intersections among wires, to listing some simple maintenance tasks to do on the flows. Concerning multiple choices questions, only one answer among the proposed was correct and counted 1 point each, while for open questions 1 point was given to totally correct answers and 0 otherwise. For each object (i.e., DiaMH and WikiDataQuerying), the questions asked to the participants were exactly the same, independently from the treatment that had occurred (i.e., non-compliant or compliant).

The participants had to complete each task in the order defined by the group they were assigned to, and to stop each task only when completed. For each task, participants had to import the corresponding Node-RED flow into Node-RED and, for each question, track start time, answer the question, and track stop time. To limit fatigue effect, each participant took a short break between the two tasks. Moreover, since the participants were trained in Node-RED during the 4-hours lectures preceding the experiment, and were split into four groups where the treated and non-treated objects were assigned in different orders, as explained by Table 6.3, we also tried to limit the learning effect.

Finally, the participants were asked to complete the post-experiment questionnaire, to collect insights about their skills and motivations for the obtained results. Questions were about the perceived complexity of the two tasks, the exercise usefulness, the feelings and the preferences between the styles of the two flows, and the competencies required to complete the tasks. Answers were provided on a Likert scale ranging from one (Strongly Agree) to five (Strongly Disagree).

6.4.7 Analysis

Because of the sample size and mostly non-normality of the data (measured with the Shapiro-Wilk test [SW65]), non-parametric test was adopted to check the three null hypotheses, accepting the customary probability of 5% of committing Type-I-error [WRH+12], i.e., rejecting the null hypothesis when it is actually true.

Since participants answered to the questions on the two different objects (DiaMH and WikiDataQuerying) with the two possible treatments (non-compliant and compliant), a paired Wilcoxon test was used to compare the effects of the two treatments on each participant.

To measure the magnitude of the effects of the two treatments, the non-parametric Cliff's delta (d) effect size [GK05] was used, which is considered small (S) for $0.148 \leq |d| < 0.33$, medium (M) for $0.33 \leq |d| < 0.474$, and large (L) for $|d| \geq 0.474$.

Table 6.4: Descriptive statistics per treatment and results of paired Wilcoxon test.
TotalComprehension is measured as the number of correct answers in a task (from 0 to 16), TotalTime is measured as the minutes required to complete a task, TotalEfficiency is the ratio of these two variables.

Dependent Variable	Non-Compliant Treatment (⁻)			Compliant Treatment (⁺)			p-value	Cliff's Delta
	Median	Mean	St. Dev.	Median	Mean	St. Dev.		
TotalComprehension	9.500	9.600	2.319	13.500	12.800	2.044	0.00903	– 0.69 (L)
TotalTime	58.500	71.100	38.484	57.000	59.100	25.291	0.04883	0.17 (S)
TotalEfficiency	0.180	0.179	0.109	0.254	0.253	0.100	0.00586	– 0.36 (M)

6.5 EXPERIMENTAL RESULTS

In this section, the effect of the main factor on the dependent variables (*TotalComprehension*, *TotalTime*, and *TotalEfficiency*), as resulted from the experiment, and the post-experiment questionnaires, are discussed.

Table 6.4 summarizes the essential *Comprehension*, *Time*, and *Efficiency* descriptive statistics (i.e., median, mean, and standard deviation) per treatment, and the results of the paired Wilcoxon analysis conducted on the data from the experiment with respect to the three dependent variables.

6.5.1 H_{0a}: Comprehension (RQ1)

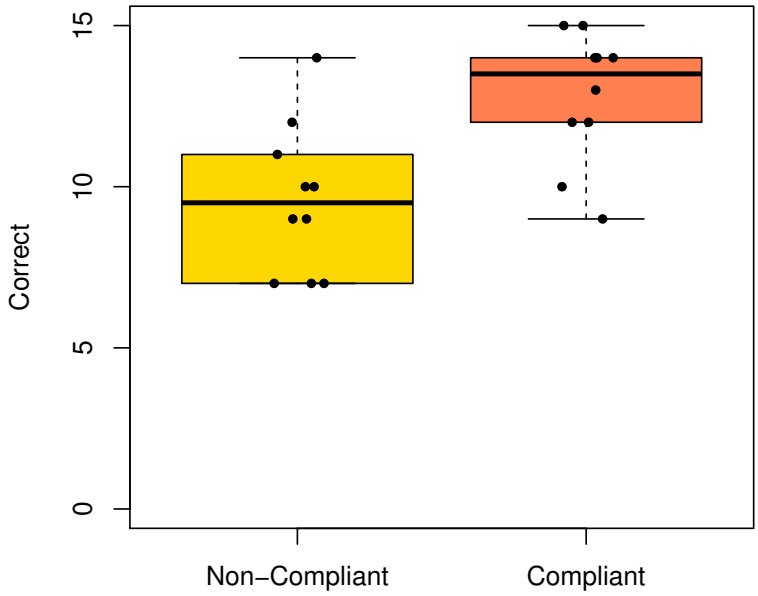
Figure 6.2 summarizes the distribution of *TotalComprehension* by means of boxplots. Observations are grouped by treatment (non-compliant or compliant). The y-axis represents the average comprehension measured as number of correct answers on the 16 questions for each treatment, where a score equals to 16 represents the maximum value of comprehension and corresponds to provide correct answers to all the 16 questions.

The boxplots show that the participants achieved a better comprehension level when working on the compliant Node-RED flows (median 13.5) with respect to those working on non-compliant flows (median 9.5).

The application of a Wilcoxon test (paired analysis) shows that the difference in terms of comprehension is statistically significant, as testified by p-value = 0.00903. Therefore, the null hypothesis H_{0a} can be rejected. The effect size is large (d = – 0.69).

To answer RQ1: The adoption of the guidelines significantly improves the level of comprehension of the Node-RED flows.

Figure 6.2: Boxplots of Comprehension.



6.5.2 H_{0b}: Time (RQ2)

Figure 6.3 summarizes the distribution of *TotalTime* by means of boxplots, where the y-axis represents the total time to answer the 16 questions for each treatment. The boxplots show that the participants needed slightly more time to answer the questions pertaining the objects with the non-compliant treatment w.r.t. those answering the questions pertaining the objects with the compliant treatment (58.5 versus 57.0 minutes respectively in the median case).

By applying a Wilcoxon test (paired analysis), it results that the overall difference is marginally significant (p-value = 0.04883). Therefore, the null hypothesis H_{0b} can be rejected. The effect size is small (d= 0.17).

To answer RQ2: The adoption of the guidelines marginally reduces the time required to answers the questions pertaining the Node-RED flows.

6.5.3 H_{0c}: Efficiency (RQ3)

Figure 6.4 summarizes the distribution of *TotalEfficiency* by means of boxplots.

The boxplots show that participants working on the objects with the compliant treatment outperformed in terms of efficiency those working with the objects with the non-compliant treatment (medians 0.254 versus 0.180, respectively).

Figure 6.3: Boxplots of Time.

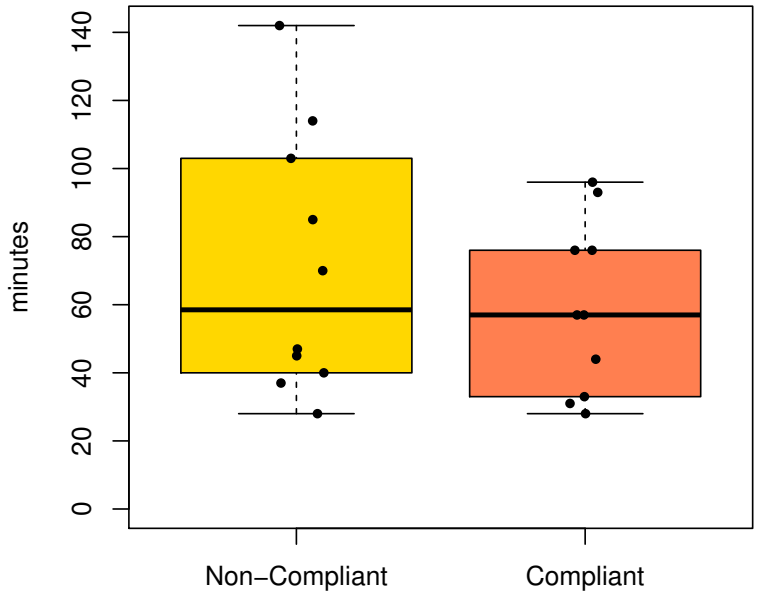
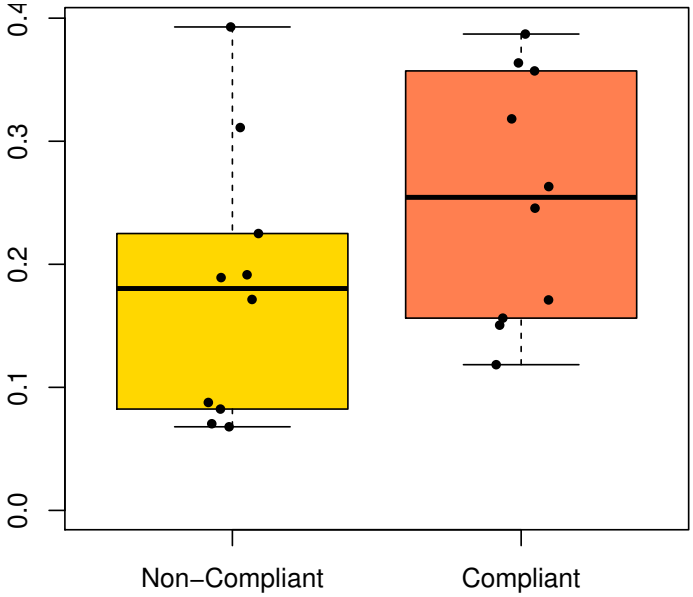


Figure 6.4: Boxplots of Efficiency.



The Wilcoxon test (paired analysis) states that the overall difference is statistically significant, as shown by the p-value (p-value = 0.00586). Therefore, the null hypothesis H_{0c} can be rejected. The effect size is medium (d = – 0.36).

To answer RQ3: The adoption of the guidelines increases the overall efficiency in the comprehension of the Node-RED flows.

6.5.4 Post Experiment

When participants had to fill the post-experiment questionnaire, they were unaware of the guidelines and, therefore, of the two treatments. For this reason, the actual questions were formulated as a comparison between the flows they had worked on in the two tasks, keeping track of which treatment occurred on them, according to the group the participants were assigned to. Thus, for instance, question **PQ1** was originally formulated as *Comprehending the Node-RED flow in Task 1 was harder than the Node-RED flow in Task 2*. In Table 6.5 the post-experiment questionnaire has been adjusted in order to clarify to the reader the purpose of the experiment. Table 6.5 reports also the medians of the answers given by the participants. The possible choices for each answer, on a 5-point Likert scale, were: Strongly Agree, Agree, Unsure, Disagree, Strongly Disagree.

Table 6.5: Adjusted post-experiment questionnaire.

ID	Question	Median
PQ1	Comprehending the non-compliant Node-RED flow was harder than the compliant one	Unsure
PQ2	In your opinion, developing the non-compliant Node-RED flow is harder than the compliant one	Agree
PQ3	In your opinion, maintaining the non-compliant Node-RED flow is harder than the compliant one	Agree
PQ4	The names of the nodes and the variables in the non-compliant Node-RED flow were less useful for the comprehension than in the compliant one	Unsure
PQ5	The wiring style to connect nodes in the non-compliant Node-RED flow was less useful for the comprehension than in the compliant one	Agree
PQ6	I found the exercise useful	Agree
PQ7	I had enough knowledge to answer the questions	Agree

As the Table 6.5 shows, participants did not perceive any difference in the complexity while trying to comprehend the Node-RED flows using each treatment (**PQ1**), but believed that developing and maintaining such flows may result more complex with the non-compliant treatment (**PQ2-3**). The names of the nodes and of the used variables in the two treatments had no significant perceived impact in the overall comprehension (**PQ4**), whereas the wiring style was better perceived in the case of the compliant treatment (**PQ5**). In general, participants found the exercise useful (**PQ6**) to the course of their studies, and fitting their knowledge in Node-RED (**PQ7**), in part acquired by attending the 4-hours lecture preceding the experiment.

6.5.5 Discussion

Given the results of the experiment, all null hypotheses can be rejected. The guidelines are generally beneficial to the comprehension level and reduce the time required to complete Node-RED tasks. Consequently, the overall efficiency is also positively affected.

One of the main reasons of success of the guidelines was producing flows that follow a consistent and tidy wiring style, by means of **WSC** and **WST**, which improved the capability of detecting loops and reduced entangles among wires. This is corroborated by **PQ5** of Table 6.5. In fact, the questions pertaining comprehensibility issues about wires and loops presented generally better outcomes for the flows compliant with the guidelines. For example, a question in both questionnaires requires to identify the number of intersections between wires. While there was only 1 error for the flows compliant with the guidelines, for the non-compliant flows the errors amounted to 8.

On the other hand, by the feeling of the participants, giving proper names to nodes was not so relevant for the comprehension (**PQ4**). This is contradicted by the results of the experiment, since the importance of names, given by **NNB** guideline, resulted to be helpful in indirectly answering several questions on both systems. For instance, there were two open questions specifically asking the names of the nodes responsible for a certain behavior (e.g., *Which node (provide name) displays on a web page the WikiData answer to the user's question?*), which resulted in a total of 4 errors for the flows compliant with the guidelines against 10 for the non-compliant ones. There were also two questions in both questionnaires asking about which data was changed/returned after the completion of a certain activity (e.g., *Which data are saved just after the HTTP request to URL?*): while there was just 1 error in the flows compliant with the guidelines, 5 errors were counted for the non-compliant cases. One question asked about the effective contributions of the nodes in a selected portion of the flows (i.e., if the nodes were able to transform the message they had received); by using **NEC** guideline, the inactive nodes (e.g., those added for debugging purposes) were removed from the flows compliant with the guidelines, but not from the non-compliant flows. In this case, while participants easily identified the contributions of the remaining nodes in the compliant flows, they failed (4 errors) in identifying the inactive ones in the non-compliant flows. Finally, there was a question asking to list all the files in the file system used by the flows, which resulted in 1 error for the flows compliant with the guidelines against 4 for the non-compliant ones. Indeed, by following the guidelines, the nodes names were formulated to make their behaviors more explicit, as well as the main used variables, hence reducing the overall errors in the comprehension, as summarized by the statistics data of Table 6.4.

From **PQ1**, participants did not have a clear opinion on which treatment was easier to comprehend, while they agreed that flows produced without following the guidelines would be harder to develop and maintain (**PQ2-3**). Concerning comprehensibility complexity, it can be speculated that the uncertainty of the participants is due to the domain of the two systems: while DiaMH presents the MQTT node (i.e., a core Node-RED node used to establish a communication from/to entities

and flows using the MQTT protocol[18]) as the most complex node, WikiDataQuerying refers to WikiData repository and to Prolog and SPARQL languages, which could deviate from the average academic background of the participants. Finally, the participants recognized that the tasks they completed did not require excessive knowledge of Node-RED and were helpful for their current/next academic studies (**PQ6-7**).

To conclude, the proposed guidelines have shown to be useful in terms of comprehension level, time, and overall efficiency. This may suggest Node-RED developers to apply them to reduce the comprehensibility issues of the flows they will produce, deploy and share. At the same time, the guidelines may turn useful to the designers of the Node-RED language, who may want to fix some of the issues here exposed, by introducing additional features in future Node-RED releases. Just to mention few possible additions: (i) nodes resizing in height and width, to highlight the most important nodes and make long names more readable, (ii) general warnings, to notify the presence of unused variables, incomplete conditions within switch nodes, and loops, and (iii) jumps between wires, to graphically handle collisions.

6.5.6 Threats to Validity

The threats to validity that could have affected the experimentation are: *internal*, *construct*, *conclusion* and *external* [WRH+12].

Internal validity threats: these threats concern factors which may affect the dependent variables. The participants had to complete two tasks; therefore, a fatigue/learning effect may have intervened. However, since they had a break between the two tasks and they previously completed some exercises about Node-RED comprehensibility issues, this effect is expected to be limited. Another threat is the subjectivity in the objects selection. The objects were flows chosen from a list of systems developed by former master students of another course related to Node-RED, and were comparable in size and complexity and composed of mostly Node-RED core nodes.

Construct validity threats: these threats concern how comprehension and time were measured. The correctness of the answers was manually checked by comparing the provided answers with the correct ones. The execution time was measured based on the time sheets filled by the participants. The statistics data (i.e., median, mean, and standard deviation) and the results of the paired Wilcoxon analysis were computed using Excel and R[19].

Conclusion validity threats: these threats concern the limited sample size of the experiment (ten master students), which may have affected the statistical tests. Unfortunately, this is the average number of students of any Computer Science Master Course in Genova, so it is generally difficult to conduct experiments with more participants.

[18] https://cookbook.nodered.org/mqtt/
[19] https://www.r-project.org/

External validity threats: these threats can limit the generalization of the results and, in this case, concern the use of students as experimental participants. The participants had few knowledge of Node-RED, therefore more expert Node-RED developers may produce a different outcome, requiring further investigations.

Chapter 7

Generation of Node-RED Flows and Test Scripts from UML-based Specifications

As introduced in Chapter 6, Node-RED is a practical solution for developing IoT systems, since it offers a large set of nodes covering a variety of functionalities. Moreover, the produced flows and nodes can be easily shared. Indeed, producing Node-RED flows that are easy to understand is fundamental to early detect faults and deviations from a system expected behavior, also to simplify the next natural maintenance and testing activities.

Testing in particular is the aim of this chapter. In fact, even if there exist rough frameworks for partially testing Node-RED nodes and flows, no fully fledged technique driving the detection of faults and deviations has emerged yet in literature.

This chapter presents a preliminary approach for developing and testing a Node-RED system starting from a UML model of its dynamic and static aspects. The JSON objects representing the Node-RED flows of the system are generated from the model, along with executable test scripts exercising selected portions of the flows.

The content of this chapter has been published in the *1st International Workshop on Ensemble-Based Software Engineering* (EnSEmble 2018) [CLRR18].

7.1 INTRODUCTION

In Chapter 6, Node-RED tool has been introduced as a practical solution for IoT systems development, with a particular focus on the comprehensibility issues that may emerge while employing it in such activity, which may affect even later steps in the process, such as testing and maintainability.

Indeed, proposing effective methods and approaches for developing and testing IoT systems is essential, but brings a number of challenges [BCA18], even when a simple tool like Node-RED is employed. An example is a flow having three nodes sequentially wired: an "inject node" to simulate an external event, like clicking on a physical button, followed by a "function node" to filter out the bad values received from the external source, followed by a "debug node" to display the good ones. This simple example can be directly tested in Node-RED by checking the values displayed by the debug node, or by using one among the nodes purposely provided by the Node-RED community for the verification of nodes and flows (e.g., assert node[1]). However, in case the example becomes trickier, it is undeniable that, due to the heterogeneity of the involved functionalities, that in some cases may employ freshly released and barely tested nodes, testing a complete flow can be hard. The Node-RED community has tried to answer to the users' testing demand[2], but even if finding online materials is relatively easy, like guides[3] and specific posts on message boards[4], neither systematic approaches nor methods have been yet proposed to test Node-RED flows.

In recent years, some proposals for assuring the quality of IoT systems [LCO+18, KAH+18] and, more generally, of Cyber-Physical Systems (CPSs) [KFK14, SZF15] have emerged, but these works do not specifically aim at supporting the user in the development and testing activities of a Node-RED system and do not provide solutions for automatically generating Node-RED flows and test scripts strictly aligned to a system requirements specification.

Therefore, corroborated by the aforementioned reasons and inspired by the Node-RED testing framework, the so called *test helper node*[5], a valid solution for testing Node-RED flows from a unit level, this chapter sketches an approach for effectively developing and testing Node-RED systems from their UML requirements specifications. From Chapter 6, it emerged that Node-RED and UML share some common points when it comes to the design of a system; therefore, given a UML requirements specification describing the desired static and dynamic properties of the system to develop in Node-RED, it should be possible to generate Node-RED flows from the UML artifacts, and associated test scripts.

The approach works as follow. First, the dynamic and static aspects of the system are modeled using UML activity and class diagrams, from which JSON objects representing the working Node-RED flows compliant to the UML model are generated. Then, by selecting portions of the model, enriched with control points to establish which properties to check, it is possible to generate test scripts able to exercise the corresponding system flows. The test scripts will be generated in Javascript, which is the core Node-RED language, and rely on Mocha[6] test framework, which runs on Node.js and supports many assertion libraries. Interactions in the early phase of the approach

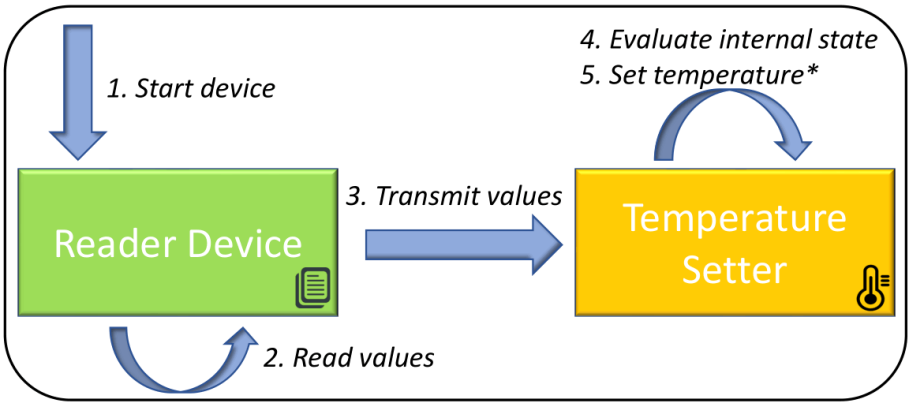
[1] https://www.npmjs.com/package/node-red-contrib-assert
[2] https://github.com/node-red/node-red/wiki/Testing
[3] http://noderedguide.com/
[4] https://discourse.nodered.org/
[5] https://www.npmjs.com/package/node-red-node-test-helper
[6] https://mochajs.org

between the professionals and the stakeholders are necessary, in order to obtain useful feedback for developing the right system without introducing faults or deviations.

In Section 7.2 of this chapter a simple running example is introduced, while the approach is outlined in Section 7.3.

7.2 THE RUNNING EXAMPLE

Figure 7.1: The running example.



The chosen running example is a simple IoT system to be developed in Node-RED, and consequently tested, sketched in Figure 7.1. The system is composed of a Reader Device and a Temperature Setter. The Reader Device reads from a continuous file stream the last 24 hours environmental degrees Celsius temperatures (range [-20, 40]) recorded outside a room by an external source. The values are transmitted to the Temperature Setter, by using the MQTT protocol, which evaluates its internal state in the following way: it computes the average of the received values and checks it against the average computed the day before and, depending on the variation between the two averages, it sets its internal state to either *Colder Temperature*, if the old average is higher than the new one and the temperature inside the room has to be increased, *Same Temperature*, if no changes are needed, or *Warmer Temperature*, if the old average is lower than the new one and the temperature inside the room has to be reduced. Depending on the internal state and on other parameters, finally the Temperature Setter sets the daily temperature inside the room.

It could be possible that the developers have not yet decided how to precisely handle each Temperature Setter state; for example, they may want it to be implemented in Node-RED as a variant of the many flows involving a thermostat node[7]. The development and the testing activities

[7] e.g., `https://www.npmjs.com/package/node-red-contrib-ramp-thermostat`

over such system should not be limited because of some unclear parts in its behavior; instead, the developers may want to have a portion immediately working as compliant, even if incomplete, and another portion to be iteratively refined and tested in the future, hence substituted by a mock that behaves in a scripted way. Notice that step *5. Set temperature** in Figure 7.1 is labeled with an asterisk to indicate that still has to be precisely defined and, for the sake of the example, will be temporary replaced by a mock.

7.3 THE APPROACH

The approach is sketched in the activity diagram of Figure 7.2.

Different tasks have to be completed in order to generate, from the system expected behavior (and associated properties), the compliant Node-RED flows and the executable test scripts to check the presence of faults or deviations. Node-RED Flows Generation and Mocha Test Script Generation tasks are marked in red because they are intended to be tool-supported.

The behavior and the properties of Node-RED IoT systems are here modeled in UML, since is widely known and used [RLR14, RLRC15] and can naturally describe the dynamic aspects of Node-RED flows, by means of activity diagrams, and the static properties of the nodes (e.g., the body of a function, the TCP communication settings), by means of class diagrams and OCL expressions. Moreover, the XML Metadata Interchange (XMI) standard adopted by UML models is supported by many tools (e.g., Papyrus[8]) and transformations to other languages based on such standard already exist (e.g., ecore.js[9] for Javascript).

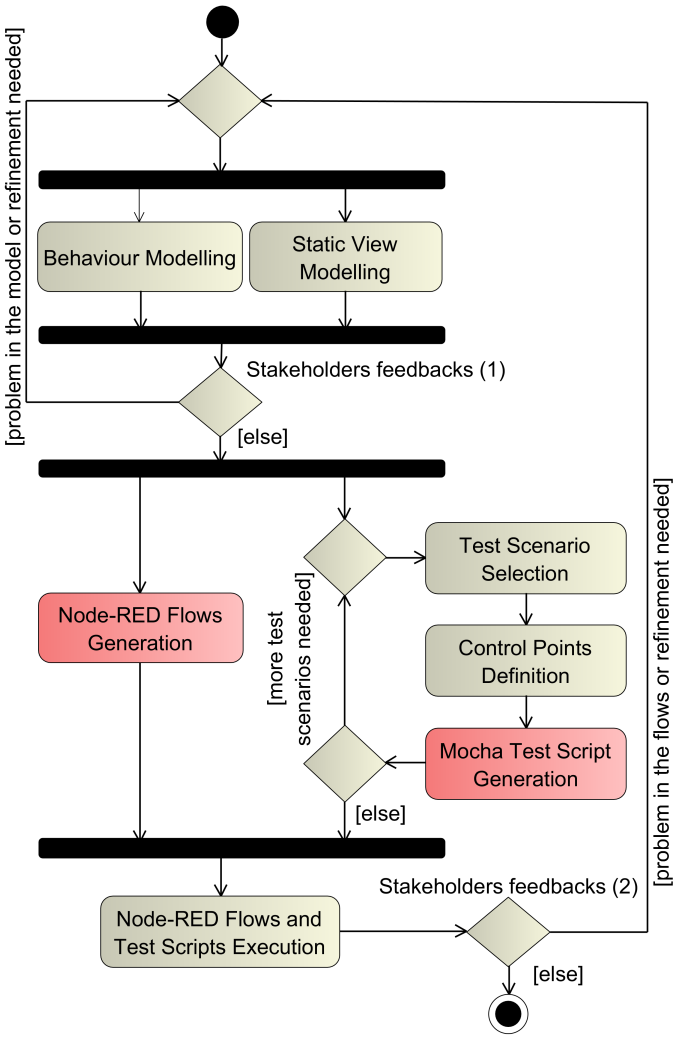
7.3.1 Behavior Modeling

This task requires the designer to model, with a UML activity diagram, the behavior of the system that is intended to be developed and tested. In this task, only the structure of the flows is important (i.e., the nodes and the wires between them). Currently, the activity diagram is restricted to the following UML constructs that are sufficient to represent the basic Node-RED nodes[10]: *action nodes, activity nodes, decision/merge nodes, fork/join nodes, object nodes, swimlanes, activity edge connectors, initial nodes, activity final nodes, flow final nodes, send signal events, accept (time) events, exception handlers, input pins, and control/object flows*.

In the approach, each Node-RED node has its own UML counterpart. For example, the *inject node*[11] transmits information based on the external event it receives, which can be repeated in time

[8] https://www.eclipse.org/papyrus/
[9] https://emfjson.github.io/ecore.js/
[10] http://noderedguide.com/node-red-lecture-3-basic-nodes-and-flows/
[11] https://nodered.org/docs/user-guide/nodes#inject

Figure 7.2: The proposed approach.



(e.g., send a message every 10 seconds), hence it is represented as a UML accept (time) event, where the event can be timed depending on its repeatability. Another example is the *switch node*, already discussed in 6.2, that in UML is represented with a decision node, since it has to handle multiple scenarios. To improve the model understandability, stereotypes representing the various nodes are added to the corresponding UML constructs.

Message passing is an activity performed by almost every Node-RED node, but in some cases the message a node returns is an untouched or a slightly changed version of the received one. Hence, while modeling the behavior of a system, only when needed to improve the understandability, messages are made explicit by means of UML object nodes exposing their properties in the form of $\{\text{property}_1\text{: value}_1, \ldots, \text{property}_N\text{: value}_N\}$, adhering to the Node-RED message JSON format.

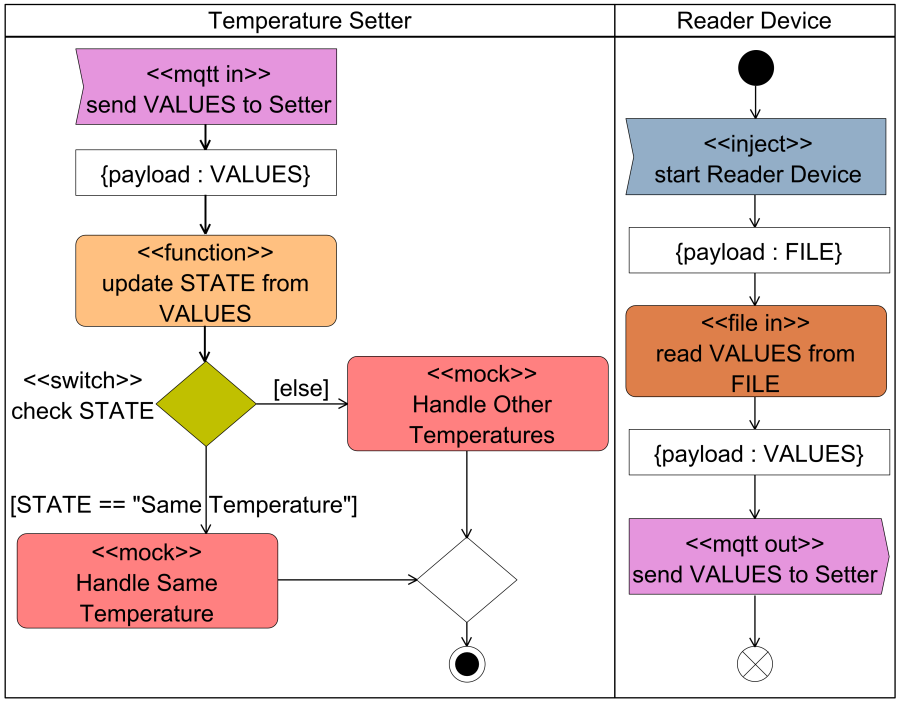
For example, if a *function node* adds a property P with value V to a received message, the returned message will be modeled as a UML object node labeled with {. . . , P:V, . . . }.

In the approach, swimlanes are used for representing the main Node-RED flows of a system. Each lane corresponds exactly to a main flow, then the placement of any UML construct representing a Node-RED node within a certain lane determines the node scope to the corresponding flow. In the running example of Figure 7.1, the system is composed of two devices, hence it will require two lanes representing its two main flows.

At this stage, modeling IoT systems in Node-RED can be tough, due to the number of heterogeneous and interconnected devices to handle and to all the technical/configuration details required by Node-RED nodes. For this reason, the approach introduces the concept of mocked portions of a system behavior. A *mocked portion* is something that the designer may not want to model yet, because of not immediate interest or because further time for thinking is required (in Figure 7.1, see step *5. Set temperature**), and then she mocks it with a scripted or a simplified behavior. A UML activity node stereotyped by ≪mock≫ is used any time a portion of the behavior of a system has to be mocked.

Figure 7.3 provides a simplified UML activity diagram representing the behavior of the running example.

Figure 7.3: The behavior of the running example.



Two lanes are used to delimit the system components: a Temperature Setter and a Reader Device. The system starts once the Reader Device receives an external event, modeled as an accept event and stereotyped with ≪inject≫. The node returns a message (a UML object node) having as payload the name of the file stream where the temperatures are stored (the variable FILE). Then, a ≪file in≫ node modeled as a UML action node reads the values from the file and returns them, again as a message payload sets to VALUES variable, to the Temperature Setter through a ≪mqtt out≫ node, modeled as a UML send signal node, and the flow ends, as shown by the UML flow final node. The Temperature Setter receives the message using a ≪mqtt in≫ node named accordingly and returns it to a ≪function≫ node, which updates the internal state (a flow variable named STATE) depending on the received VALUES; no details about the function have to be given in the activity diagram, since they will be provided in the static view (see Figure 7.2). Then, a ≪switch≫ node receives the message from the function node and checks the value of STATE, resulting in two possible activities: Handle Same Temperature and Handle Other Temperatures. These two activities are modeled as mocked portions of the system, as shown by their stereotypes, which means that they are not yet intended to be developed and will present a scripted behavior, specified in the static view, without blocking the execution or the testing of the system.

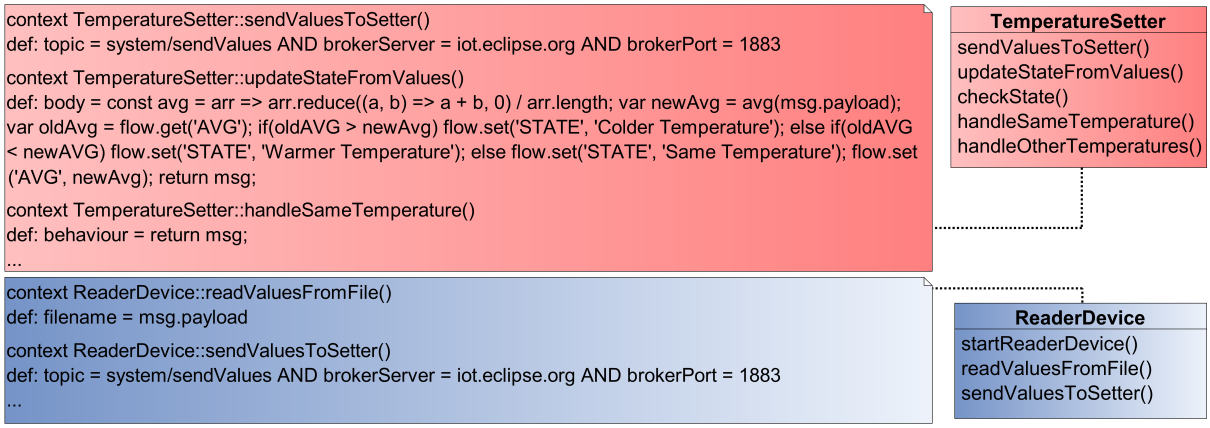
From the example, it is notable that not all the UML constructs correspond to Node-RED nodes; indeed, the constructs with neither colors nor stereotypes are just used by UML for modeling purposes. For example, the UML flow final node at the end of the Reader Device lane states when the flow ends, but has no equivalent representation in Node-RED. Similarly, the UML merge node of the Temperature Setter lane is used only for merging together the two alternatives exiting from the previous UML decision node. More generally, there is not a bijective correspondence between the UML constructs in the activity diagram and the nodes in the Node-RED flows; in fact, the two notations present different syntax and semantics. In any case, UML includes all the information needed to generate basic Node-RED flows from a UML model.

7.3.2 Static View Modeling

This task is conducted in parallel with the Behavior Modeling task, see Figure 7.2, and requires the designers to model, with a UML class diagram, the static view of the system that is intended to be developed and tested.

More specifically, the class diagram exposes classes, named *flow classes*, each one representing a lane of the activity diagram (i.e., the main Node-RED flows of the system). A flow class contains an operation for each UML construct representing a Node-RED node included in the corresponding lane. Finally, OCL notes are attached to each flow class to define their operations, i.e., the properties of the nodes, formulated as a conjunction of Property = Value. The definition of the operations does not require parameters or returned values. Indeed, most of Node-RED nodes receive messages and return messages, sometimes changing their structures, hence making messages passing explicit would not add any information; instead, whenever a message has to

Figure 7.4: A portion of the static view of the running example.



be made explicit, it is modeled in the activity diagram as a UML object node exposing all the interesting properties, as shown in Figure 7.3.

The approach requires iterative steps for modeling the behavior and the static view of a system, see the loops in the process of Figure 7.2, hence the definition of some nodes properties which may result too complex at this early stage can be postponed, for instance those requiring precise technical details (e.g., the server name and the port number of a communication node) or average programming skills (e.g., the Javascript body of a function node). A partial static view, defining some properties of the nodes of the activity diagram modeled in Figure 7.3, is shown in Figure 7.4.

Since the activity diagram is composed of two lanes, corresponding to the Reader Device and the Temperature Setter main flows of the system, the static view presents two flow classes, each one having an attached OCL note defining its operations. For instance, sendValuesToSetter of the TemperatureSetter class represents the homonym MQTT node receiving the values from the Reader Device, hence it requires communication properties such as the topic (i.e., basically, the message identifier), the broker's server and the broker's port (i.e., basically, the configuration of the server routing the messages). Instead, updateStateFromValues of the same class represents the homonym function node and has a property named body which embodies its logics in Javascript language, i.e., *it computes the average of the received values and compares it with the one computed the day before, stored in a flow variable named AVG. Depending on which average is higher, it changes a flow variable named STATE that will be used for the next temperature setting*. Notice also the definition of handleSameTemperature; the operation represents the homonym mocked portion of the system and its behavior is defined to simply returning the message it receives, hence doing nothing. This definition will have to be changed once that mocked portion of the system is clearer.

7.3.3 Stakeholders Feedback

The approach requires strong interactions between the stakeholders and the professional figures responsible for modeling, developing and testing the system, as shown in Figure 7.2. Indeed, since having Node-RED flows aligned with the system model is a mandatory requirement of the approach, it is imperative that the stakeholders can analyze the produced artifacts and provide feedback. This can happen in two phases: one at the beginning of the process (Stakeholders feedback (1)), when a problem in the model is identified or a refinement of the model is needed (e.g., update the class diagram by defining a new node property or update the activity diagram by modifying a flow), and one at the end of the process (Stakeholders feedback (2)), when a problem in the generated Node-RED flows or in the test scripts is identified or when a refinement of the model is needed (e.g., some mocked portions have to be defined or a node property has to be fixed to address a fault).

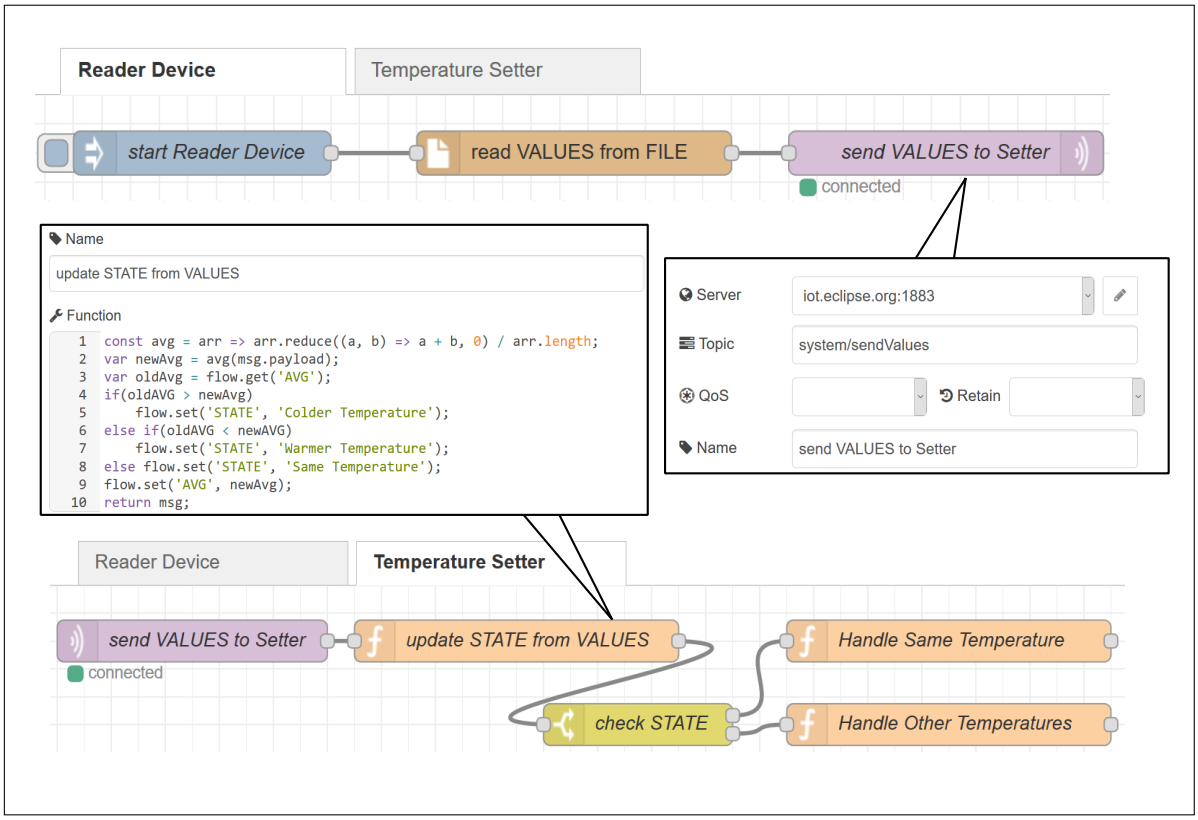
7.3.4 Node-RED Flows Generation

Once the system has been modeled, it is possible to transform the produced UML model into Node-RED artifacts, by generating the flows and the nodes properties from the activity and the class diagrams, respectively. Basically, it is a transformation from XMI (for the UML perspective) to JSON (for the Node-RED perspective), which is intended to be automated. The transformation will apply the following sketched procedure:

- For each lane L_i in the activity diagram, generate an empty Node-RED flow F_i;
- For each UML construct C_k in lane L_i, if C_k is stereotyped as S_k, generate a Node-RED node N_k in flow F_i having the form {id: n_k, name: C_k, type: S_k, z: L_i, wires: [ ]}, where z is the property that Node-RED uses to identify a flow;
- For each flow class FC_i in the class diagram, for each operation O_{ki}, add the definition of O_{ki}, having the form {$property_1$ = $value_1$, . . . , $property_N$ = $value_N$} to the properties of the node N_k in flow F_i, changing = with :;
- For each couple of UML constructs C_n and C_m in lane L_i, having stereotypes S_n and S_m respectively, if there exist a sequence of transitions from C_n to C_m such that no other stereotyped UML construct is in the sequence, and if S_n permits output wires and S_m permits input wires, then add n_m to the property wires of node N_n in flow F_i.

The first step generates empty flows from the UML swimlanes; the second step generates for each stereotyped UML construct a corresponding empty node with predefined properties, i.e., id, name, type, associated flow, and nodes the current node is wired with (initially none); the third step adds

Figure 7.5: The generated Node-RED JSON flows and nodes properties of the running example.



to each node the properties defined by the corresponding operation in the static view; the final step adds a wire between two nodes if they should be connected.

A possible outcome of the procedure is given in Figure 7.5, where the Node-RED flows of the Reader Device and the Temperature Setter components of the running example are generated from the activity and the class diagrams shown in Figures 7.3 and 7.4. Notice the mocked portions transformed into function nodes, each one having a scripted behavior.

7.3.5 Test Scenarios Selection

In the approach, the testing activity over the system is performed in parallel with the generation of the Node-RED flows, as shown in Figure 7.2. The produced activity diagram describes the complete system behavior, including mocked portions, therefore test scenarios can be selected from it.

A *test scenario* is defined as a physically or a logically connected portion of the system behavior, composed of several UML constructs. The connection between UML constructs is essential for

having a test scenario: it is physical when two UML constructs C1 and C2 are directly connected with a UML control/object flow; it is logical when C1 and C2 logically represent a direct sequence of related events, for instance if C1 is a UML send signal node stereotyped with ≪mqtt out≫ and C2 is a UML accept event node stereotyped with ≪mqtt in≫.

Since a test scenario is a partial view of the system behavior, it does not represent a fully working Node-RED flow, hence it may lack preceding steps where variables and message properties are set. Then, each test scenario has to be preceded by a *tailored mocked portion*, responsible for setting the used global and flow variables and the intended message properties. As the name suggests, each tailored mocked portion is tailored to a specific test scenario. For instance, we may want to check that a component A can send a message M to a component B and, based on the payload of M, B can execute either sub-flows S1, S2 or S3; hence, depending on the way the payload is set by the tailored mocked portion defined for that test scenario, one among those three sub-flows may be exercised.

As for the mocked portions introduced while modeling the behavior of a system, even the tailored mocked portions of test scenarios are represented as UML activity constructs stereotyped with ≪mock≫, in this case preceding the first UML constructs of the test scenarios they are associated with and placed in special lanes (e.g., Testing lane). Moreover, each tailored mocked portion has to be defined by adding to the class diagram, used for modeling the static view of the system, a new class named as the newly introduced lane, that will represent the tailored mocked portion as an operation defined in an OCL note in the form of behavior = B, where B is expressed in Javascript.

At this stage of the approach, a precise strategy for selecting the best test scenarios, in terms of system coverage, and the smartest way for customizing the tailored mocked portions, still require to be properly defined. Both topics will be investigated in the next future.

7.3.6 Control Points Definition

To proceed in the testing activity of the system, the selected test scenarios must be completed by adding some control points. In the approach, a *control point* corresponds to any assertion formulated over a system property, i.e., a global/flow variable or a message property, and is represented as a UML action node stereotyped with ≪control point≫. The idea of adding control points within Node-RED flows has been inspired by some of the Node-RED nodes and frameworks having verification purposes, e.g., the *assert node* and the *test helper node* mentioned in the introduction of this chapter, which can be added within a Node-RED flow to intercept the information passed among the observed nodes.

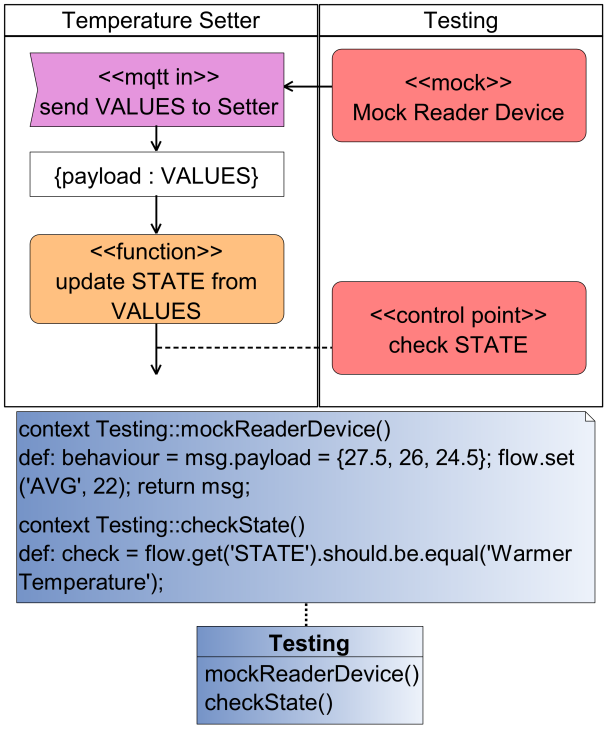
Each control point attached to a test scenario has to be added to the same lane of the tailored mocked portion for that test scenario and its definition is given in an OCL note attached to the

flow class in the class diagram corresponding to that lane, in the form of check = C, where C is expressed in Javascript.

Theoretically, checks defined by control points could be formulated relying on the plethora of libraries and modules running in Javascript and Node.js (e.g., Should[12], Chai[13], Assert[14]). Examples of checks using the Should library are var.should.be.equal(V), where var is a global/flow variable or a message property and V is a primitive value, and, msg.should.have.property(P, V), where P is the name of a property that a message msg should have and V is a primitive value. More complex checks over system properties will be investigated, e.g., checks over the time taken during a communication, comparisons of the output produced by multiple executions of the same test scenario, and more.

Figure 7.6 shows, on top, a test scenario selected from the system behavior modeled in Figure 7.3.

Figure 7.6: A test scenario (top) and tailored mocked portion and control point definitions (bottom) of the running example.



Since the test scenario focuses on just a portion of the Temperature Setter lane, a tailored mocked portion named Mock Reader Device is introduced at the beginning of the test scenario and added to a new lane named Testing. The test scenario ends after the function node, ignoring what

[12] https://shouldjs.github.io/
[13] http://www.chaijs.com/api/assert/
[14] https://nodejs.org/api/assert.html

happens next, therefore a control point is attached to the transition exiting from it. On the bottom of Figure 7.6, the class representing the Testing lane is shown, including the note defining both the tailored mocked portion and the control point. The tailored mocked portion is defined by feeding the test scenario with a message payload of three temperatures (27.5, 26, and 24.5), to simulate the values the Reader Device reads from the file stream, and by setting a flow variable named AVG to 22, to simulate the average computed the day before. This means that the new average temperature computed by the function node of the Temperature Setter (see its definition in Figures 7.4 and 7.5) over the three received values is 26, slightly higher than the one stored in AVG, hence the value of the STATE variable after the function node should be equal to Warmer Temperature, as expected in the definition of the control point named check STATE.

7.3.7 Mocha Test Scripts Generation

Once the control points are introduced in a test scenario and consequently defined, it is possible to generate the corresponding test script, relying on the Mocha test framework. The generation of the test scripts is intended to be automatically conducted by a tool that will apply the following sketched procedure. Given a test scenario TS:

- Generate an empty Mocha test script MTS;
- Generate a JSON flow F from TS and declare it in MTS;
- Extract from TS the unique stereotypes S representing the Node-RED nodes and declare them in MTS. This step relies on the require[15] Node.js built-in function to load modules;
- Add a load function LF to load F in MTS;
- Find the tailored mocked portion TMP from F and declare it in LF;
- Find the control points $CP_1 \ldots CP_n$ from F and declare them in LF;
- For each control point CP_i, add a check instruction CI_i in LF;
- Add the instruction to start F with TMP at the end of LF.

A simplified test script generated by the aforementioned procedure, which corresponds to the test scenario shown in Figure 7.6, should look like Listing 7.1.

[15] https://nodejs.org/api/modules.html#modules_require_id

Listing 7.1: The generated Mocha test script corresponding to the test scenario.

```
1  var helper = require("node-red-node-test-helper");
2  var mqttInNode = require("./node_modules/node-red/nodes/core/io/10-mqtt.js");
3  var functionNode = require("./node_modules/node-red/nodes/core/core/80-function.js");
4  it("Test Scenario 1", function(done) {
5    var flow = [
6    {id: "n0", name: "Mock Reader Device", type: "function", func:"msg.payload = {27.5, 26,
       24.5}; flow.set('AVG', 22); return msg;", ..., wires:[["n1"]]},
7    {id: "n1", name: "send VALUES to Setter", type: "mqtt in", ..., wires:[["n2"]]},
8    {id: "n2", name: "update STATE from VALUES", type: "function", ..., wires:[["n3"]]},
9    {id: "n3", name: "check STATE", ..., type: "helper"}
10   ];
11   var nodesTypes = [functionNode, mqttInNode];
12   helper.load(nodesTypes, flow, function () {
13     var mock = helper.getNode("n0");
14     var cp = helper.getNode("n3");
15     cp.on("input", function(msg){
16       cp.context().flow.get("STATE").should.be.equal("Warmer Temperature");
17       done();
18     });
19     mock.receive();
20   });
21 });
```

Lines 1-3 are built-in functions referencing to the nodes that appear in the flow; notice, in particular, the *test helper node* (line 1) needed by Mocha to find the nodes within the flow and to load the flow in the test environment. The function it (line 4) represents a test script in the Mocha environment. Inside the test script, there are the declarations of the flow (lines 5-10) and of the nodes types included in the flow (line 11); some properties have been hidden for space purpose (refer to Figure 7.4 for a better understanding). Notice the presence of the tailored mocked portion at the beginning of the flow (line 6) and of the control point at the end (line 9); in particular, the control point is of type helper referring to the *test helper* module, since it provides a useful interface for easily recovering information from flows and nodes it observes. Once the flow is loaded, after the callback (line 12), both the tailored mocked portion and the control point are extracted from it, by using their identifiers (lines 13-14), then the control point waits for input events from the preceding function node (line 15) and, once received the message, checks the value of the flow variable STATE (line 16), as defined in Figure 7.6. The last instruction in the load callback (line 19) activates the flow, by means of the tailored mocked portion simulating the reception of a message to be returned to the next node.

Chapter 8

An Acceptance Testing Approach of IoT Systems

Day by day, IoT systems are becoming ubiquitous for human activities, therefore assuring their quality is fundamental. Unfortunately, few proposals for testing IoT systems are present in the literature, in particular when the entity to test is the system as a whole.

This chapter presents an acceptance testing approach of IoT systems adopting graphical user interfaces as the principal way of interaction. Acceptance testing is a type of black box testing based on test scenarios, i.e., sequences of steps/actions performed by the user or the system.

In the approach, the development and testing activities are driven by a UML state machine that expresses the core behavior of the target IoT system. To evaluate its effectiveness, the approach has been applied on a realistic case study, a mobile health IoT system for diabetes management composed of heterogeneous devices, and compared against an existing runtime verification approach in terms of strengths and weaknesses. Results have shown the effectiveness of the approach, which was able to detect between 71% to 100% of the types of bugs injected in the system.

The content of this chapter has been published in the *Journal of Information and Software Technology* (IET Software 2018) [LCO+18] and in the *14th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE 2019) [LCF+19].

8.1 INTRODUCTION

As the IoT technology continues to mature, we will see more and more novel IoT systems emerging in different contexts. For example, trains able to dynamically compute and report arrival

times to waiting passengers, cars able to avoid traffic-jam by proposing alternative paths, and m-health systems able to determine the right medicament dose for a patient.

The importance of assuring the quality of IoT systems is therefore undeniable. As emerged in Chapter 7, testing these kinds of systems can be difficult even when a simple tool like Node-RED is employed, due to the wide set of disparate technologies used to build them (hardware and software) and the added complexity that comes with Big Data (the three "V" [ZE+11], huge volume, great velocity and big variety).

In Chapter 7, a preliminary proposal for testing IoT systems developed in Node-RED was introduced. However, the proposal is strictly based on Node-RED and operates from a unit level of testing. This chapter instead presents a black-box testing approach of IoT systems that works from an acceptance level. The approach is oriented to systems presenting a GUI as the principal way of interaction, and is based on three main steps: (i) IoT system behavior formalization, (ii) IoT system development and virtualization, and, (iii) testing artifacts generation. The system behavior is formalized by means of a UML state machine (or multiple state machines) describing the states of the system as changes occurring in the GUI due to internal/external events. Then, the system is developed in Node-RED for the core logics parts, following the behavior described by the state machine; in case several complex devices are employed, mocks and emulators can be introduced to virtualize part of the system behavior. Last step aims at generating test scripts exercising Node-RED flows from the paths of the UML state machine. For test artifacts generation, different testing frameworks to drive the GUI could be employed. The current approach consider two of them: *Appium*[1], a testing framework relying on structural-based localization of the GUI components, and *SikuliX*[2], a testing framework based on image-recognition localization. The testing artifacts are based on the concept of *test scenario*, i.e., a sequence of actions performed on the system GUI, which is considered by many organizations [PAR16] the most effective way to ensure the quality of a fully deployed system. In fact, assembling an IoT system and testing it as a whole is the starting, most logical, and effective way to ensure its quality.

The effectiveness of the approach has been evaluated by taking, as case study, a GUI-equipped mobile health IoT system for the management of diabetic patients, composed of a sensor, an actuator, a cloud-based healthcare system, and smartphones. The evaluation starts, first, by generating a set of mutated versions of the system using *Stryker*[3] tool (in this way, a large number of possible bugs that a developer could introduce can be simulated, both during development and maintenance activities), and, second, by executing both test suites (Appium and Sikuli) against each mutant, noting down if at least a test script was able to detect the erroneous behavior and, thus, the corresponding mutation in the source code. A thorough discussion on the comparison of the adopted testing frameworks is provided, based on the experience gained on the case study, that could help practitioners having to face similar testing tasks in choosing which tool suits better.

[1] `http://www.appium.io/`
[2] `http://sikulix.com/`
[3] `https://stryker-mutator.github.io/`

Finally, the approach has been compared against a runtime verification approach developed by some of the authors that contributed to the testing approach outlined in this chapter, for better understanding their strengths and weaknesses of the two strategies.

The chapter is organized as follow. Section 8.2 presents the case study, then Section 8.3 describes the approach applied on the case study, Section 8.4 describes the empirical study evaluating the approach with respect to the two different testing frameworks, and, finally, Section 8.5 compares the testing approach with the runtime verification approach and discusses the results.

8.2 THE CASE STUDY

As case study, a diabetes mobile health IoT system was chosen for three reasons. First, these kind of systems are incredibly difficult to test and no consolidated approaches have been proposed for them in literature. Second, many software apps for smartphones and IoT systems for patients are now available and intended to assist them in making decisions for themselves in real time [Klo13]. Finally, the acceptance testing approach presented in this chapter, where the GUI is exercised in terms of its functionalities as a final user would do, may result useful considering the number of apps for diabetes and general healthcare systems based on active human participation [IKK+15]. The proliferation of such apps and systems is due to the fact that diabetes is a very common disease doubling a person's risk of early death. Just to give two estimates [WHO16]: 422 million people have diabetes worldwide (2016) and the World Health Organization (WHO) reports that diabetes resulted in 1.5 million deaths in 2012, making it the 8th leading cause of death [WHO13]. Insulin therapy is often an important part of diabetes treatment and is injected to the patient to keep the blood glucose level low.

8.2.1 DiaMH - A Diabetes Mobile Health IoT System

DiaMH is a Diabetes Mobile Health IoT system that: (1) monitors the patient's glucose level, (2) sends alarms to the patient and the doctor when a glucose level trend is out of a pre-specified target range, and (3) regulates insulin dosing. DiaMH, sketched in Figure 8.1, consists of the following components: a wearable glucose sensor, a wearable insulin pump, a patient's smartphone, a doctor's smartphone, and a cloud-based healthcare system. Glucose sensor and insulin pump are devices (respectively, the sensor and the actuator) connected to the smartphone, that is used as a "bridge" between them and the cloud-based healthcare system. Moreover, the smartphone is used by the patient and by the doctor to visualize details and notifications pertaining the patient's health. The cloud-based healthcare system is the core of DiaMH and is able to process big data and turn it into valuable information (alarms and novel doses of insulin).

Figure 8.1: Components and actors of DiaMH.

Notice that a modified version of DiaMH system was given as a students project and developed by former Computer Science master students of the University of Genova, and chosen as object of study in the work discussed in Chapter 6 about the Node-RED guidelines.

A thorough testing phase is required, since DiaMH is a complex, real-time, and safety-critical IoT system. Currently, there are no well-documented approaches that can be used for acceptance testing of a system like DiaMH. Indeed, only very general non-scientific papers concerning testing of IoT systems[4] and several proposals for testing bioinformatics software (e.g., [CHLX09]) are available. Unfortunately, these works cannot be directly used for assuring the quality of DiaMH, since they do not describe a specific testing solution.

The ingredients of the approach are: formalization of the system expected behavior in UML, development and virtualization of physical sensors/actuators through mocks in Node-RED, and test artifacts generation implemented using acceptance test automation tools/frameworks. The main advantage of test automation comes from fast, unattended execution of large sets of tests after some changes have been made to the system under test. Test automation tools can automatically report the results to developers, and compare them with earlier test runs. The focus is on test automation tools/frameworks [LCRT16] that control the execution of test scripts giving commands directly on the GUI (e.g., the smartphone interface) and reading actual outcomes to be compared with predicted outcomes. Mock devices are pieces of software that mimic the behavior of real devices/systems in controlled ways (e.g., the glucose sensor providing glucose profiles of different kinds of diabetics patients); mocks are fundamental for testing the system without dealing with physical devices. Thus, the approach requires to build a set of acceptance test scripts that, when run, can drive the DiaMH system execution by giving commands on the smartphone GUI, as a real patient does, and verifying the correctness of the DiaMH system through the GUI interface.

[4] e.g., https://devops.com/functional-testing-iot/

8.2.2 DiaMH Functionalities, Components and Protocols

This section specifies the functionalities of DiaMH and clarifies the roles of the components used for implementing it. The following description has been inspired by two works: a Parasoft white paper [PAR16] and the paper by Istepanian *et al.* [IHPS11].

The ***glucose sensor*** (e.g., SugarBeat®[5]) is a pain-free, non-invasive, needle-free sensor, that monitors the blood for detecting the glucose level, and performs measurements in timed intervals (sampling rate). The intervals can be specified in the DiaMH settings and delivered by the patient's ***smartphone*** with a specific command. The glucose sensor is wirelessly connected to DiaMH app on the patient's smartphone. The app stores the values locally and provides a GUI so that the patient can monitor the glucose levels in the blood and the messages describing the current status. Also, the app running on the patient's smartphone transmits glucose level data to a cloud-based healthcare system for determining the patient's status, the insulin doses, and persistent storage.

The ***insulin pump*** follows a programmed schedule controlled wirelessly by the patient's smartphone. Smartphone, glucose sensor and insulin pump use the Bluetooth Low energy technology [GOP12] to communicate among them. TCP is the protocol used in the communication between smartphones, sensors, actuators, and healthcare system.

The ***healthcare system*** running on the cloud is the core of the DiaMH IoT system. It stores the data of all the patients, compares them to historical data, performs advanced analyses, computes the insulin quantities that should be injected by the insulin pump and sends alarms to the doctor and patient in case of danger, i.e., when problematic patterns are identified. To perform the analyses and control the insulin pump, the healthcare system uses cognitive computing, machine learning algorithms and, in general, artificial intelligence mechanisms. Indeed, establishing the "correct" dose of insulin is incredibly complex, since the insulin requirements are affected by the individual's physiology, the type and duration of daily activity, work schedule, exercise, illness and concomitant medications [MR16]. The alarm sent to the patient contains some data, such as a guidance for the next steps and changes to the insulin dose (if any) to be delivered to the insulin pump. When a problematic pattern is identified, the healthcare system sends an alarm directly to the doctor's app. The alarm sent to the doctor contains the GPS coordinates of the patient and a summary of her medical conditions.

8.2.3 Acceptance Testing on Virtualized/Real IoT Systems

Testing the aforementioned system poses significant challenges, given that DiaMH is composed of several components working together and with risk of individual failure. Moreover, further problems could derive by the integration of the components. Indeed, it is well-known that an "imperfect" integration can introduce a myriad of subtle faults.

[5] http://www.nemauramedical.com/sugarbeat/

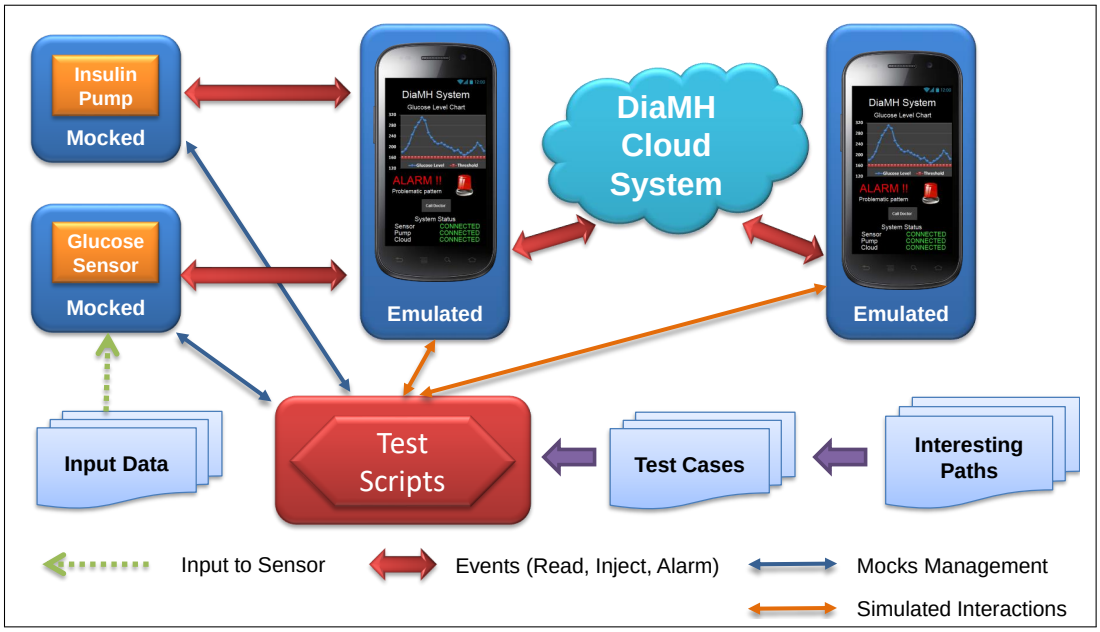
In general, a complete test plan should include a combination of unit testing (components should be isolated and tested early), integration testing (components should be tested as a group, proceeding, e.g., bottom-up), and acceptance testing, and should be conducted at two different levels:

1. testing a ***virtualized version of DiaMH system***, where real hardware devices are not employed. In their place, virtual devices (e.g., a mocked glucose sensor, an emulated smartphone) have to be implemented and used for stimulating the applications under test. At this level the goal is testing only the software developed for DiaMH, i.e., the apps (for patients and doctors) running on the smartphones, the logics managing the communication between the various components and the healthcare system running in the cloud. Thus, possible unwanted behaviors of DiaMH due to hardware or physical network problems cannot be detected at this level.
2. testing the ***real DiaMH system***, complete of applications and devices (i.e., glucose sensor and insulin pump). The goal here is testing the system in real conditions, i.e., under real world scenarios like communication of the application with hardware, network, and other applications.

Since DiaMH is safety-critical, both testing levels should be conducted because the former could favor earlier implementation problems detection and could potentially reveal more faults (timings of sensors and actuators can be made shorter and thus a huge quantity of tests can be executed in a short time, for instance by setting the sampling rate at 1 Hz instead of 15 minutes). In this chapter, the focus is on testing level (1) because it is the first one that a test team has to face and can be conducted without employing real sensors and actuators that could be expensive and complex to use/set. Moreover, since devices used in m-health systems are usually certified, they are not the main reason of failure of the entire system. The approach is oriented to acceptance testing because this phase seems to be neglected more than the others in the IoT domain, since the specific peculiarities of testing an IoT system (e.g., composed by different platforms, relying on various communication protocols, and including complex behaviors of smart-devices) are more evident when testing the whole system than the single, isolated components.

Figure 8.2 reports an overview of the elements of DiaMH involved in the approach: the mocked glucose sensor and insulin pump, the emulated smartphones where the app is running, the testware (i.e., the interesting paths extracted from the UML state machine representing the system behavior, the abstract test cases, and the corresponding executable test scripts), and the healthcare cloud system.

Figure 8.2: Elements involved in the Acceptance Testing Approach.



8.3 THE APPROACH

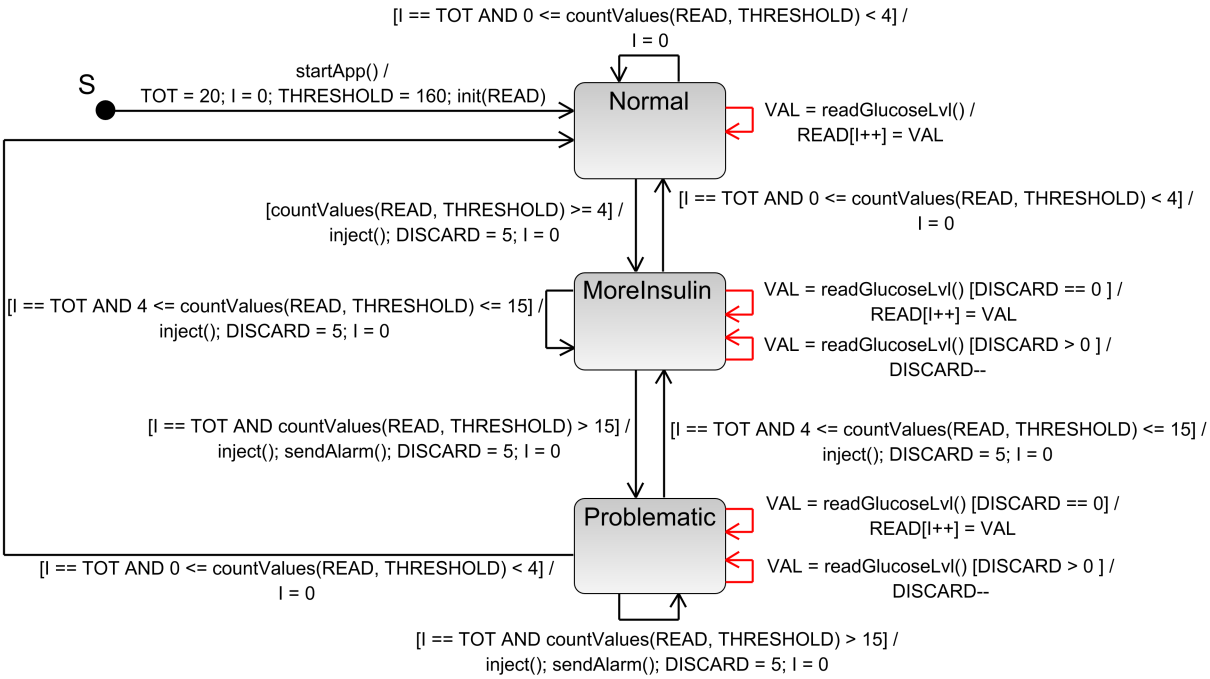
In the following, the approach is applied on DiaMH case study, from formalization to test scripts implementation.

8.3.1 DiaMH System Behavior Formalization

The approach focuses on the acceptance level perspective, hence a precise description of the expected system behavior is required. A possible choice is to formalize it in terms of a UML state machine (or multiple state machines) to guide the testing activities, as suggested by various model-based testing techniques where state-based models describing systems behaviors are used for code generation and test cases derivation [UL10]. Figure 8.3 formalizes the expected behavior of the core part of DiaMH, i.e., the logic that recognizes the status of the patient and decides when to administer an insulin dose, manages the glucose sensor and the insulin pump, and allows to show the information to the patient's smartphone. The black transitions lead to DiaMH GUI state updates while the red ones are used for managing the incoming data from the glucose sensor. In the following, a slightly simplified version of DiaMH is considered: (1) the part of DiaMH dealing with the doctor's app is discarded, since it is similar to the patient's app; and, (2) the healthcare cloud system is considered as a deterministic system with a precise and repeatable behavior (e.g., at same glucose levels must correspond exactly the same insulin prescriptions).

Thanks to these simplifications, the test cases can be enriched with very strict assertions that allow a careful evaluation of the quality of the DiaMH implementation under controlled conditions. Indeed, evaluating the behavior of DiaMH including the results provided by complex analysis based on big datasets and performed by machine/deep learning algorithms (whose results may also vary over time, as in case of online learning [SS12]) would require to apply techniques like, for instance, metamorphic testing [SFSC16], in order to cope with partially unpredictable behaviors/values. That would introduce an additional confounding factor to the evaluation of the effectiveness of the approach; for this reason, a deterministic version of the healthcare cloud system is considered. Notice that each transition in the state machine is composed of trigger [guard] / actions; the trigger is an optional event which activates the transition it is associated with (e.g., startApp()), the guard is a boolean expression which must be true to enable the associated transition (e.g., countValues(READ, THRESHOLD) $\geq$ 4), and the actions are responses to the trigger activating the associated transition (e.g., READ[i++] = VAL).

Figure 8.3: DiaMH core expected behavior.



In the UML state machine of Figure 8.3, there are three states representing a generic patient's condition: *Normal*, *More Insulin*, and *Problematic*. When the DiaMH is bootstrapped (startApp()), the initial state is set to *Normal* and the glucose threshold (THRESHOLD) discriminating good values from bad ones is set to 160 mg/dl. In the following, the DiaMH sampling rate is assumed to be of 1 Hz. Thus, each second, the glucose level is read by the sensor (readGlucoseLvl() in the red transitions) and stored in a 20 elements circular buffer (READ). If there are no more than 3 values in the buffer above the pre-set threshold (computed by countValues(READ, THRESHOLD)),

the patient is considered stable and remains in the *Normal* state. Otherwise, if 4 or more values are above the threshold, a *More Insulin* pattern is detected. In this case, a new insulin dose is injected by the pump (inject()) and the state machine switches to *More Insulin* state, while the next 5 readings are discarded (the counter decremented after each discard is DISCARD), since the injected insulin dose will need time to take effect, which is hypothetically estimated in 5 time units (i.e., 5 seconds when the sampling rate is 1 Hz). From this state, if the next 20 readings are between 4 and 15 values above the threshold, the patient is kept in a *More Insulin* state, a new insulin dose is injected and the process is repeated. Otherwise, if the dangerous values are no more than 3, the patient is considered stabilized and the state machine goes back to the *Normal* state. Finally, in case there are more than 15 values above the threshold, a *Problematic* pattern is detected and the state machine moves accordingly, just after injecting a new insulin dose and notifying the problem by sending alarms to the patient's smartphone (sendAlarm()). Again, from the *Problematic* state, the next 5 readings are discarded; then, after 20 further readings, the state machine can stay in the same state, injecting a new dose and sending new alarms, can move back to *More Insulin* after an injection, or can even move to *Normal*.

Once the behavior has been properly formalized by means of a UML state machine (for a safety-critical system, like DiaMH, such behavior specifications is assumed to be available even before starting the system development phase), the next stages are:

- The development in Node-RED of the core system logics, the development in Java of the app running on the smartphones, and the virtualization in Node-RED of the devices;
- The generation of the testing artifacts (composed of *definition* of the test scenarios and input data, and *implementation* of the test scripts).

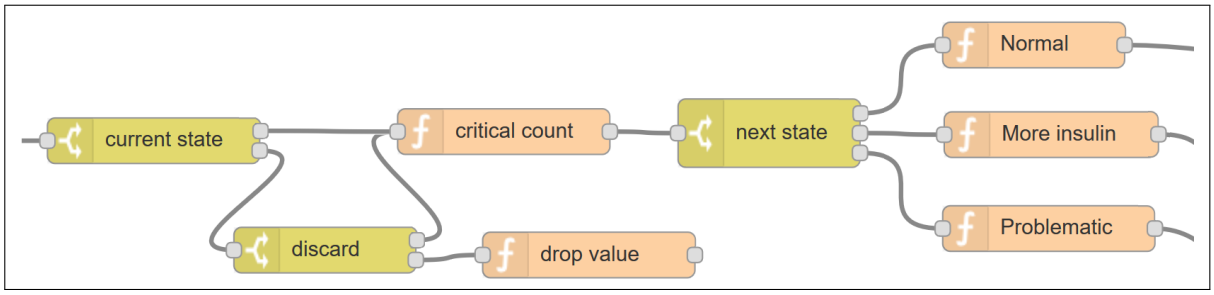
8.3.2 DiaMH System Development and Virtualization

The development of the core parts of DiaMH is based on Node-RED, in particular the deterministic version of the healthcare cloud system, described in the UML state machine of Figure 8.3, and the communications between the various components, using the TCP protocol, as well as the virtualization through mocks of the glucose sensor and the insulin pump. Indeed, Node-RED is expressly designed for wiring together hardware devices, APIs and online services, therefore can simulate even complex devices behaviors, by using even simple JavaScript function nodes. In this way, it is possible to provide the DiaMH system with selected input patterns and evaluate its capability in sending commands to the devices (e.g., perform an insulin injection) and dealing with their messages (e.g., acknowledge of performed injection).

Instead, the development of the application GUI and the emulation of the patient's and doctor's smartphones where the application will run are based on Java with Android Studio[6] and Android

[6] https://developer.android.com/studio

Figure 8.4: A portion of DiaMH healthcare cloud system in Node-RED.



Emulator[7], respectively; the emulator will allow to execute and test the DiaMH mobile apps from a computer workspace.

A portion of DiaMH healthcare cloud system implementation in Node-RED is given in Figure 8.4. The current state switch node at the beginning of the flow uses a global variable to determine the current patient's state; at the starting of the system, such variable is set to *Normal*, and then it is updated periodically as described in the specifications. In case the current state is *Normal*, the flow continues to critical count function node, which reads the glucose values from the sensor and counts those above the fixed threshold of 160 mg/dl. Instead, in case the current state is *More Insulin* or *Problematic*, the flow moves to discard switch node, which verifies that the next 5 readings following a pump injection are discarded; until some values have to be discarded, the flow moves to drop value function node, which drops the next 5 readings from the sensor, otherwise it reaches critical count function node, counting the readings above the threshold, as mentioned above. From critical count node, the flow goes to next state switch node, to determine the patient's next state after the readings from the current state, by checking the amount of critical values: from 0 to 3 the flow moves to *Normal* node, from 4 to 15 to *More Insulin* node, and above 15 to *Problematic* node. Each one of these function nodes implements, in Javascript, a portion of the behavior described in Figure 8.3 concerning the associated state.

8.3.3 Test Scenarios and Test Cases Definition

A *test case* is a list of actions performed on the DiaMH system, and one or more assertions evaluated over properties that can be graphically checked. Test cases have to exercise all the *interesting scenarios* described by the UML state machine, where the triggers, the guards and the actions in the transitions determine the instructions of the test cases, while the assertions are formulated considering what may graphically be displayed during a change of state (also in case of a self-loop), e.g., *is an alarm displayed on screen any time a Problematic state is reached?*.

[7] https://developer.android.com/studio/run/emulator.html

In the following, the activity is limited to test the black transitions between different states in DiaMH, instead of testing that every single value from the glucose sensor is correctly received. Including the red transitions (see Figure 8.3) would produce by far more detailed test cases, for instance with an assertion for each value coming from the glucose sensor. This would allow to perform a low level, detailed testing of the communication with the sensor, but, on the other hand, would require to: (1) have real-time GUI updates showing the last value received from the glucose sensor, but such level of details may result confusing to the patients using the app, (2) define each single value of each pattern and their exact order, since the assertions would have to check them. The cost of such a detailed level of testing is high; instead, the approach tries to determine if a larger grain and lower cost testing process (i.e., considering only the black transitions of Figure 8.3) is still capable of detecting problems in the implementation.

For defining the paths and thus the test scenarios from which test cases can be derived, several coverage criteria could be considered (e.g., node, transition and path coverage). In the case of DiaMH: node coverage leads to just one path (i.e., from starting the app (S) to Problematic) and thus to a single test scenario, which is insufficient for any testing purposes; transition coverage leads to two paths (i.e., from S to Problematic and directly to Normal, and, from S to Problematic and back to Normal - via More Insulin), plus the paths for the self-loops, but in this way it is not possible to create a test case for each state-change; finally, path coverage is infeasible given the presence of loops. For this reason, an ad-hoc path coverage criterion was applied to allow the definition of all the *interesting scenarios*, considering only the black transitions (see Figure 8.3). The rationale behind defining the *interesting scenarios* is the following: without considering the self-loops, it is required at least a test case ending with an assertion for each possible state in the state machine; if a state can be reached through different transitions, a test case ending in this state for each incoming transition is required; moreover, if a state can be reached by a transition but following different combinations of states, then a test case for each combination is required as well; finally, for each self-loop in the state machine, it is required a test case that, as last step before the final assertion, follows such loop. To reach this goals, an algorithm that works as follows was conceived:

- find the length k of the longest path(s) that can be found from the starting state to any other reachable state, without considering self-loops and without traversing the same transition in a single path twice. In the case of DiaMH, it is: from starting the app (S) to Problematic and back to Normal (via More Insulin), which requires to traverse five transitions ($k = 5$).
- generate all paths of length $l = 1..k$ from the starting state S to any other reachable state, without considering self-loops and without traversing the same transition in a single path twice. In the case of DiaMH, considering only the behavior of the core part of DiaMH, the number of paths amounts to 7.
  - $l = 1$: from starting the app (S) to Normal
  - $l = 2$: from S to More Insulin

- $l = 3$: from S to Problematic, and from S to More Insulin and back to Normal
- $l = 4$: from S to Problematic and back to More Insulin, and from S to Problematic and directly to Normal
- $l = 5$: from S to Problematic and back to Normal (via More Insulin).

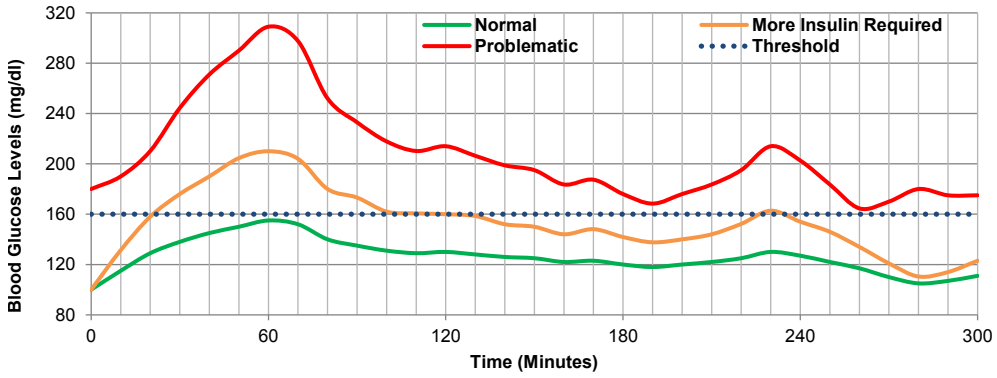
- add the shortest paths required to test the self-loops: from S to self-loop to Normal, from S to self-loop to More Insulin, and from S to self-loop to Problematic.

In this way, ten paths that correspond to ten test scenarios are found, covering all nodes, all transitions and an interesting subset of the possible paths. At this point, for obtaining actual executable test cases from these test scenarios, assertions and input data must be introduced.

Concerning assertions, at least one has to be added after the traversal of each transition in the paths defined above, where each assertion has to check properties that can be graphically visible, e.g., the aforementioned alarm in case a *Problematic* state is reached.

Having realistic input data is fundamental in order to test the DiaMH system under certain conditions. The input data will be simulated by the mocked glucose sensor taking values from log files containing real glucose patterns recorded from various kind of patients, like those plotted in Figure 8.5.

Figure 8.5: Glucose patterns: Normal, More Insulin, and Problematic.

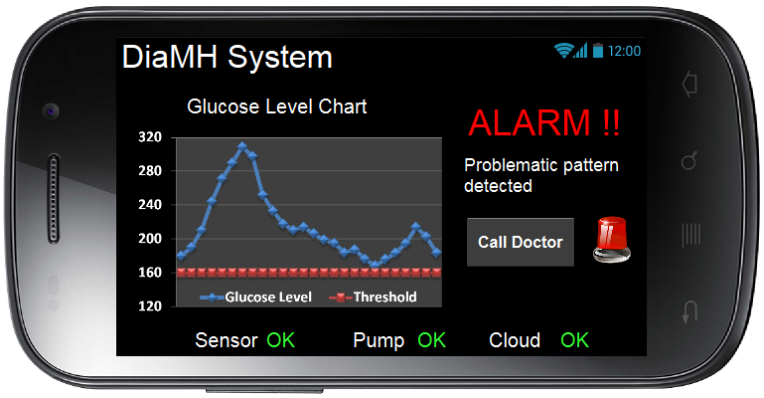


For instance, from S To Problematic test case covers the scenario of having a *Problematic* patient, from the start of the DiaMH system (S) to first 4 readings of values above the fixed threshold of 160 mg/dl (*More Insulin* state), after which an insulin injection is performed and 5 further glucose readings are discarded, to a second complete readings of 20 values, where at least 15 are above the aforementioned threshold and hence an alarm is displayed on the smartphone screen of the patient (*Problematic* state). Such test case can be described as follow (the DiaMH expected actions are also shown, in order to make clearer the description; timings t are defined by analyzing the formalization of the behavior given in Figure 8.3):

- TEST **sets** the scenario (threshold to 160 mg/dl, number of readings to 20, mock glucose sensor sampling rate at 1 read/sec, mock glucose sensor to *Problematic* pattern modality, i.e., more than 15 values above the threshold) and **sends** a reset signal to DiaMH
- DiaMH **displays** the state *Normal* in the GUI
- TEST **asserts**@$t0$ that the app has started (i.e., the state is *Normal*)
- DiaMH **reads** values from the mocked glucose sensor
- DiaMH after 4 readings above the threshold, **injects** the insulin dose and **displays** the state *More Insulin* in the GUI
- TEST **asserts**@$t4$ that the insulin dose has been injected (i.e., the state is *More Insulin*)
- DiaMH **discards** next 5 readings
- DiaMH **reads** values from the mocked glucose sensor
- DiaMH after 20 readings where more than 15 are above the threshold, **sends** an alarm
- TEST **asserts**@$t29$ that an alarm is displayed on the patient's smartphone (i.e., the state is *Problematic* after 4 + 5 + 20 reads)

Timing t is reported here without considering the actual time required to execute the commands, send the messages and refresh the GUI. Thus, in the implementation of the test scripts an "ϵ" amount of time will be added to deal with such delays. Figure 8.6 shows an example of what should be displayed on the smartphone screen in case a *Problematic* pattern is detected by the DiaMH system (i.e., the interface of the DiaMH patient's mobile app at the end of from S To Problematic test case, when the final assertion will be evaluated).

Figure 8.6: Smartphone GUI of the DiaMH app when a *Problematic* pattern is detected.



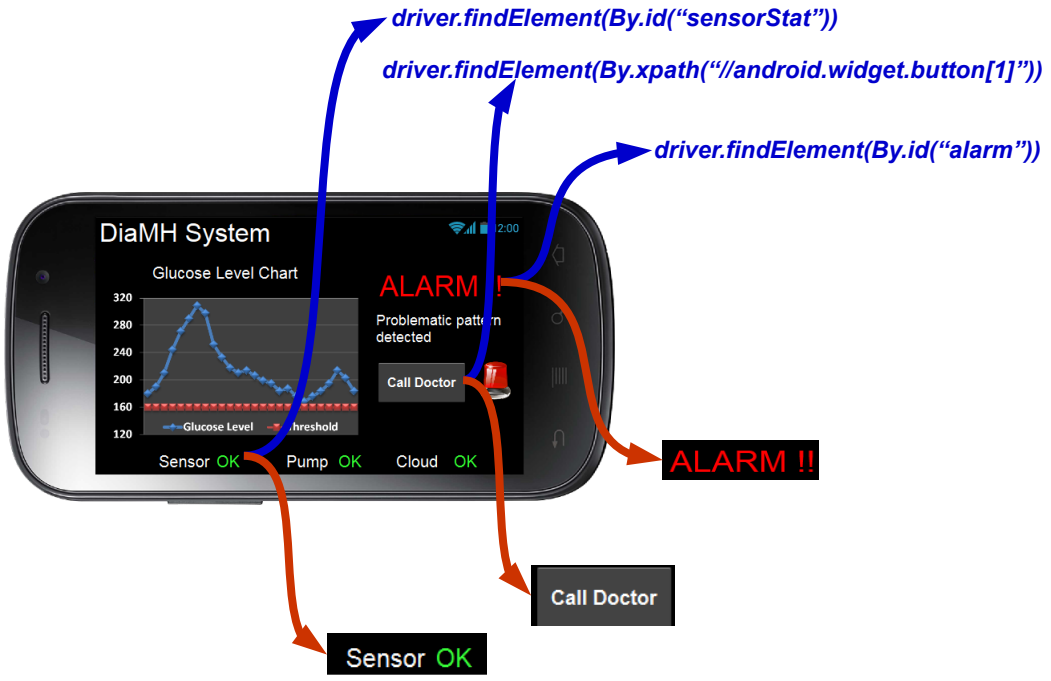
8.3.4 Test Scripts Implementation

A *test script* is the implementation of a test case (i.e., an executable test case). Hence, a test script consists of a list of commands/instructions based on the selected tool/framework and able to localize and interact with the GUI components, completed with assertions.

Currently, for IoT/Mobile systems there exist two main techniques for GUI elements localization: *visual-based* (where GUI elements are located using image recognition techniques), and *structure-based* (where GUI elements are located using the information contained in the structure of the GUI and a locator is, for instance, an XPath expression [LSRT16]). In Figure 8.7, some visual-based and structure-based locators of a smartphone GUI are shown. The red arrows indicate the relationships between GUI elements and visual locators, which are images representing portion of the GUI. The blue arrows, instead, show how the properties of the GUI (e.g., identifiers, names) or expressions (e.g., XPath expressions), are used as locators to hook to the target GUI elements.

In the example, visual-based and structure-based locators refer respectively to Sikuli and Appium test automation frameworks. Sikuli is developed as an open-source research project of the MIT User Interface Design Group, and is able to automate and test graphical user interfaces using screenshot images providing image-based GUI automation functionalities to Java programmers.

Figure 8.7: GUI elements localization: Visual-based (red) and Structure-based (blue) techniques.



Appium is again open-source and supports test automation for native, hybrid and mobile web apps and is based on Selendroid[8] and Selenium libraries[9].

Both kinds of frameworks, visual-based (with Sikuli) and structure-based (with Appium), have been employed in the approach in order to evaluate the advantages and the disadvantages while using them for testing mobile apps in the context of IoT systems. Two equivalent Java test suites composed of 10 test scripts each have been implemented, corresponding to the aforementioned test cases, where each test script in both frameworks interacts with the same GUI elements and checks the same properties. The approach is dependent from the GUI, since instructions and assertions must exercise the visible/interactive elements on the smartphone screen. However, the approach is able to test the majority of the apps running on mobile devices and emulators. Moreover, the test scripts could be simply extended to directly retrieve information from the actuators (for instance, the insulin pump mock in DiaMH), but this retrieval extension could be infeasible when dealing with real actuators (for instance, if no logging features are available). For this reason, assertions based on actions/values received directly by the mock (using, for instance, a probe or debug API) were avoided.

Figure 8.8 reports a simplified dual Sikuli/Appium test script implementing from S To Problematic test case. The smartphone app interface is encapsulated by one Java class as suggested by the *Page Object* pattern[10], where each class represents the corresponding GUI page elements as a series of class attributes and its features as class methods. Since the test scenario is decoupled from the implementation, an improvement in reusability, readability and maintainability of the test suites can be achieved [LCRT13].

Similarly, the behaviors of the glucose sensor and the insulin pump are encapsulated in classes that contain methods implementing the commands required to execute specific actions on the mocks. For instance, the method setPatternTo (see Figure 8.8) provides the mocked glucose sensor with a glucose pattern (in the example, *Problematic*), recorded in an input file.

In the example, three assertions are formulated, each one relying on visual-based (red) or structure-based (blue) locators, depending by the framework adopted for the test script implementation. The first assertion verifies that the app has been correctly loaded and that is the *Normal* state, the second one verifies that the insulin dose injection is properly displayed on screen (i.e., *More Insulin* state), while the last one verifies that the alarm message is displayed as expected in the *Problematic* state. Concerning timings, the test scripts rely on a timer that is able to assess the time elapsed since the application was started. The aforementioned ϵ delays (see Section 8.3.3) in timings can be fine-tuned and minimized for each wait or set to a fixed amount of time (i.e., $\epsilon = 0.5$ sec in the example, which is greater than the maximum delay time required in DiaMH). This does not create problems since the sampling time is set to 1 read/sec, thus with a precision of 0.5 sec it is certain that no state changes can happen in the meanwhile.

[8] http://selendroid.io/

[9] https://www.selenium.dev/documentation/en/webdriver/

[10] http://martinfowler.com/bliki/PageObject.html

Figure 8.8: from S To Problematic test script.

```
public void from_S_To_Problematic(){
   commonFunction.init(); // set the threshold to 160 mg/dl and number of readings to 20
   DiaMH app = new DiaMH();
   Timer t = new Timer();
   GlucoseSensor gs = new GlucoseSensor();
   gs.setSamplingRate(1);  // set glucose sensor sampling rate to 1 read/sec
   gs.setPatternTo("Problematic"); // set mock glucose sensor to “problematic pattern modality”
   double E = 0.5; //set the timing delay
   // start the timer and the app
   t.start();
   app.start();
   // at time 0 assert that the app has started
   assertTrue(app.isStateNormal("Normal"));
   // at time 4 after the start of the app
   // assert that the insulin dose has been injected
   t.waitElapsedTimeFromStartIs(4 + E);
   assertTrue(app.isStateMoreInsulin("More-ins"));
   // at time 29 after the start of the app
   // assert that the alarm is displayed on the patient’s smartphone
   t.waitElapsedTimeFromStartIs(29 + E);
   assertTrue(app.isStateProblematic("Problematic"));
}
```


It is worth noting that Sikuli and Appium can be also used for simulating other kinds of interactions with the GUI (e.g., clicking a button or typing in a field) but, for the sake of simplicity, the example in Figure 8.8 shows only simple visual-based or structure-based assertions.

In total, the size of the Appium test suite for DiaMH (computed using CLOC v1.74[11]) is of 524 LOC, while the size of the Sikuli test suite is of 513 LOC.

8.4 EXPERIMENTAL EVALUATION

This section reports the design, the object, the research questions, the metrics, the procedure, the results, and the discussion of the experimental study conducted to evaluate the effectiveness of the approach.

8.4.1 Study Design

The *goal* of the empirical evaluation is to investigate the effectiveness of the test suites, created by following the approach and implemented using two different testing frameworks (i.e., Appium and Sikuli), in revealing bugs in GUI-equipped IoT systems. The results of this study are interpreted

[11] http://cloc.sourceforge.net/

according to two *perspectives*: (1) *project managers*, interested in understanding which testing framework could lead to detecting more bugs; (2) *researchers*, interested in empirical data about the impact of different testing framework on IoT systems testing. The *context* of the study is defined as follows: three *human subjects* have been involved, a Postdoc, a PhD student and a Master student; the *software object* is the aforementioned DiaMH.

8.4.2 Research Questions

The experimental evaluation aims at answering the following research questions.

RQ1: *What is the number of bugs that can be detected by adopting the approach instantiated with two different testing frameworks?*

The goal is to quantify and compare the capability of the approach in revealing bugs when the test suites are implemented with the Appium and Sikuli testing frameworks. Moreover, in this way it is possible to understand if any difference exists between the two frameworks. This would give developers and project managers an idea of the bug-detection capability of test suites created by following the approach. The technique adopted for evaluating the effectiveness of the approach is *mutation testing*, while the metrics used to answer this research question is the percentage of killed mutants out of the total.

RQ2: *Is there a relation between test cases complexity and mutants detection?*

The goal of this research question is to investigate the relation between complexity of the test cases and their capability in detecting mutants. The complexity of the test cases is measured in terms of the number of transitions covered in the state machine, while the mutants detection capability is simply measured as the number of killed mutants. To answer this research question, the Pearson correlation coefficient is computed between the number of transitions covered and the number of detected mutants.

RQ3: *Is there a relation between the completeness of the GUI and the capability of the test cases in detecting mutants?*

The goal of this research question is to investigate the relation between the completeness of the information shown in the GUI (that potentially cannot express completely the expected behavior described in the UML state machine), and the capability of the test cases in detecting mutants. The completeness of the GUI is measured as the number of states that the GUI can display out of the total number of states reported by the UML state machine describing the behavior of the system, while the capability of the test cases in detecting mutants is measured as the number of killed mutants. To answer this research question, the number of killed mutants is measured on two variants of the DiaMH system presenting different GUI completeness levels.

8.4.3 Mutation Testing

Mutation testing [OU01] is a testing technique that is traditionally used for evaluating the quality of the produced test scripts, by exercising them against slight variations of the original code simulating the errors a developer could introduce during development and maintenance activities. These variations, named mutants, can identify the weaknesses in the test artifacts by determining the parts of a software that are badly or never tested [KTL15].

The idea is the following: for each mutant, the test scripts are run. A test script is effective w.r.t. a mutant if it kills the mutant, i.e., it can detect the change in the system behavior introduced by the mutant. Otherwise, the test script passes and the mutant survives, proving the test script weakness in exercising such portion of the code. The goal is to kill the highest number of generated mutants. *A measure to evaluate the overall test suite quality is given by the percentage of mutants killed over the total: the higher the better.*

The mutation phase is usually driven by mutation operators which affect small portions of code, exploiting some typical programming mistakes, like a change in a logical operator (e.g., AND instead of OR), a boolean substitution (e.g., from true to false), or a conditional removal (e.g., a conditional statement is set to true). Manually generating the mutants in a realistic scenario is clearly infeasible (and, in the context of an experiment, possibly biased), but there exist automated tools providing operators for generating a large number of mutants starting from the original code, like PIT[12] and Jumble[13] for Java, Stryker[14] for Javascript and Infection[15] for PHP.

8.4.4 Procedure

The first step in the procedure was selecting the proper mutation tool.

Among the options, Stryker tool was chosen, since it is a Javascript mutator, then suited for systems developed using Node-RED, like DiaMH, where its core logics (i.e., the deterministic healthcare cloud system, the back end of the patient's smartphone application and the entire portion managing the communication among the various components and the mock devices) is implemented in Javascript function nodes and placed in various Node-RED flows (see Figure 8.4). Stryker supports various mutant operators and plugins, and is largely configurable to properly generate and store the mutated code. It offers mutation operators for unary, binary, logical and update instructions, boolean substitutions, conditional removals, arrays declarations, and block statements removals.

To answer **RQ1**, the procedure was the following:

[12] http://pitest.org/
[13] http://jumble.sourceforge.net/
[14] https://stryker-mutator.github.io/
[15] https://github.com/infection

1. From the original code in Node-RED implementing the core of the DiaMH system, by using an automated script, all the function nodes embodying Javascript were selected and Stryker was applied on them using all the supported mutators, resulting in 39 mutants.

2. Then, a script to automatically and separately inject each mutated function node into the original Node-RED flows was implemented, resulting in 39 mutated versions of DiaMH.

3. Finally, both Appium and Sikuli test suites were run against each mutated version of DiaMH, noting down: (i) whether the mutant was killed, and if so, by which test scripts, and, (ii) a detailed analysis to explain why each mutant was killed or not.

To answer **RQ2**, the number of transitions covered in the state machine by each test case was measured.

For **RQ3**, starting from the original version of DiaMH that shows a different GUI state for each possible node in the state machine, a simplified version of the DiaMH GUI was conceived (e.g., to be used on a simple smart watch). The new GUI shows only two states out of three: (1) *Normal*, depicted as an "OK" string reported in green, and, (2) *More Insulin*, depicted as an "Injection" string reported in red representing both the *More insulin* and *Problematic* states. Since the state machine is not changed (the internal behavior of DiaMH is unaffected by this GUI change), the test cases remain the same unless the way the assertions are defined: e.g., instead of asserting either being in a *More Insulin* or a *Problematic* state, as done in the test cases for the original version of DiaMH, in the simplified version of the system the assertions only check the presence of the unique "Injection" string.

The releases of the software used in the experiment are: Appium 1.7.1, Sikuli 1.1.1, Node-RED 0.17.5, Striker 0.10.3, and Eclipse Oxygen 4.7.1a.

8.4.5 Results

RQ1: Table 8.1 summarizes the number of mutants killed by each Appium/Sikuli test script. In both cases, the same number of mutants are killed (27 out of 39), hence there are no differences between Appium and Sikuli. This is due to the fact that the test scripts are basically equivalent, excluding the way the GUI elements are located (see Figure 8.8).

Among the generated 39 mutants, 12 outlived both Appium and Sikuli test suites. However, after analyzing each outliving mutant from a code perspective, 10 out of 12 were identified as exactly equivalent to the original system behavior [JH11], hence there could not exist an acceptance test script able to kill them.

As an example of equivalent mutants, given that in Javascript some logical operators can be used also for assignments (e.g., `expr = expr1 || expr2` returns `expr1` if it can be evaluated to

Table 8.1: Mutants killed for each Appium/Sikuli test script.

Test Script	Mutants Killed	
	Appium	Sikuli
from starting the app (S) to Normal	1	1
from S to More Insulin	8	8
from S to Problematic	17	17
from S to More Insulin and back to Normal	12	12
from S to Problematic and directly to Normal	21	21
from S to Problematic and back to More Insulin	19	19
from S to Problematic and back to Normal (via More Insulin)	22	22
from S to self-loop to Normal	3	3
from S to self-loop to More Insulin	15	15
from S to self-loop to Problematic	17	17
Total Mutants killed - (a)	**27**	**27**
Total number of Mutants	**39**	**39**
Total number of Mutants (excluding equivalent) - (b)	**29**	**29**
Mutants detection rate - (a/b)	**93%**	**93%**

true, expr2 otherwise, while expr = expr1 && expr2 returns expr1 if it can be evaluated to false, expr2 otherwise), there was a mutant changing a logical operator used for an assignment from context.global.get('threshold') || 160 to context.global.get('threshold') && 160. Since the value of context.global.get('threshold'), that is the global variable storing the threshold to discriminate good from bad glucose values, was originally set to 160 and never changed, the assignment returned 160 independently from the mutation.

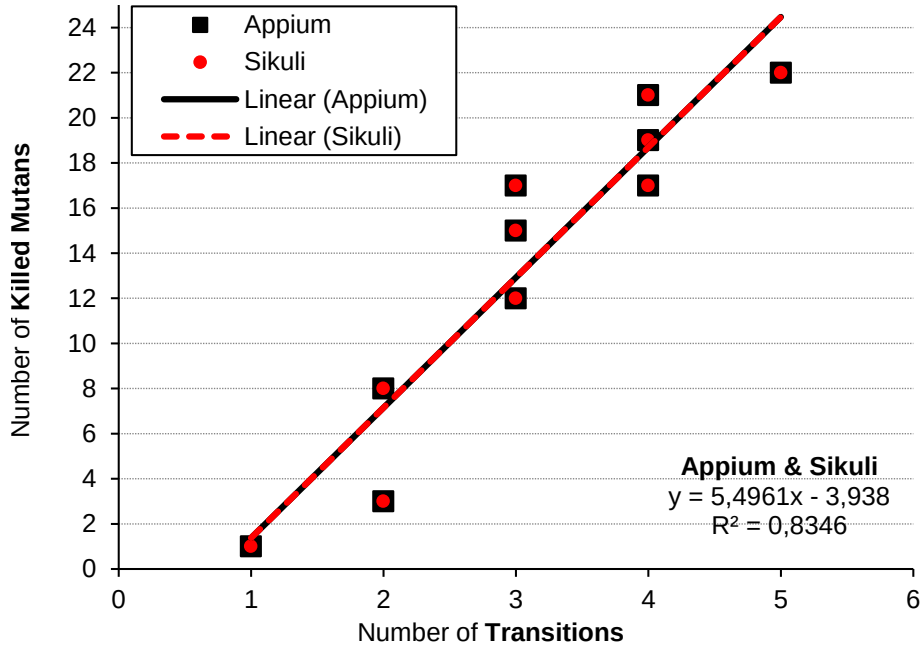
Thus, only two mutants were considered as real survivors. From the analysis, it was discovered that this happened because of weaknesses in the provided input, since the dataset was not complete enough to cover all possible conditions and properly exercise the boundaries of the original system. For example, one of these mutant changed a binary operator from if (glucose_level $>$ threshold) to if (glucose_level $\geq$ threshold). In this case, if the input file representing the condition of a normal patient had contained multiple times the exact threshold value (i.e., 160), such data would have been wrongly treated and recognized as a part of a *More Insulin* pattern, resulting in an unexpected insulin dose injection detected by some test scripts (e.g., from S to self-loop to Normal).

It is interesting to note that the test scripts developed by following the approach were able to detect mutants (i.e., bugs) located in every portion of the system.

RQ1: To summarize, the obtained results concerning mutants detection show the effectiveness of the test scripts implemented by following the approach. Indeed, if the equivalent outliving mutants (10) are excluded from the total (39), only two survived, hence the 93% of the generated mutants that actually modified the behavior of DiaMH were killed.

RQ2: looking at Table 8.1, it is interesting to notice that all the test scripts involving a *Problematic* pattern are able to kill a higher number of mutants. This can be trivially explained by Figure 8.3: to reach a *Problematic* state, it is necessary to traverse a higher number of transitions, hence any test script involving a *Problematic* pattern would exercise a larger portion of the system and would similarly perform a higher number of instructions, which may potentially lead to detect more mutants than a simpler and shorter test script could do. For the same reason, two test scripts showed a very low mutants detection rate: from starting the app (S) to Normal and from S to self-loop to Normal. For instance, in the former case, the lower mutants detection is explainable by the test script simplicity: it verifies that DiaMH is running and after a certain amount of time it checks that the patient is still in a *Normal* condition. Figure 8.9 is a scatter plot that displays the results concerning three variables for each test script: the number of transitions traversed, the number of mutants killed, and the adopted testing framework. A linear relationship emerges (R^2 is close to 1 for both Appium and Sikuli, indicating that the regression lines fit well the data) and the Pearson correlation index confirms a strong positive correlation between the number of transitions traversed and the number of mutants killed: 0.942 for both Appium and Sikuli.

Figure 8.9: Test cases complexities and mutants detection.



RQ2: To summarize, the obtained results show that high complexity in test cases, measured as number of different transitions traversed in the state machine, correspond to high mutants detection rates (Pearson correlation coefficient ≈ 0.94 for both the considered test suites).

RQ3: as shown in Table 8.2, some differences in mutants detection appear when testing the simplified version of DiaMH w.r.t. the results concerning the original DiaMH (see Table 8.1). These differences depend on the test scripts involving *More Insulin* and *Problematic* states of DiaMH, that in the simplified version are treated as a single state, hence resulting in less effective test scripts with respect to mutants affecting the behavior involving these two states. For example, from S to Problematic and directly to Normal test script killed 21 mutants in the original version and 17 in the simplified one; such result is due to the fact that some mutants had changed the check responsible for discriminating between *More Insulin* and *Problematic* states, associating instead a patient having a *More Insulin* condition to a *Problematic* condition, with no perceived differences in the simplified version of DiaMH, since the GUI was limited just to the "Injection" string and could not detect the mutated behavior. For this reason, in the simplified version less mutants were killed (i.e., 21 instead of 27).

Table 8.2: Mutants killed for each Appium/Sikuli test script for a simplified DiaMH system GUI.

Test Script	Mutants Killed	
	Appium	Sikuli
from starting the app (S) to Normal	1	1
from S to More Insulin	8	8
from S to Problematic	13	13
from S to More Insulin and back to Normal	12	12
from S to Problematic and directly to Normal	17	17
from S to Problematic and back to More Insulin	12	12
from S to Problematic and back to Normal (via More Insulin)	15	15
from S to self-loop to Normal	3	3
from S to self-loop to More Insulin	11	11
from S to self-loop to Problematic	13	13
Total Mutants killed - (a)	**21**	**21**
Total number of Mutants	**39**	**39**
Total number of Mutants (excluding equivalent) - (b)	**29**	**29**
Mutants detection rate - (a/b)	**72%**	**72%**

RQ3: To summarize, the obtained results show that there is a relation between different GUI completeness levels and the capability in detecting mutants. In fact, the more the GUI is complete w.r.t. the expected behavior of the system, by covering the majority of the states, the more the capability in detecting mutants increases. While in the original case the GUI covered all three DiaMH states and resulted in 93% of mutants killed, in the simplified version, where the GUI collapsed two states (i.e., *More Insulin* and *Problematic*) into one, the mutants detection was reduced to 72% even while adopting exactly the same test scripts.

8.4.6 Threats to Validity

In the following, some of the most relevant threats to validity of the present study are listed.

- *Authors' bias* threat. To limit this threat, the main tasks of the study were executed, each one, by a different author of the work described in this chapter: (i) DiaMH expected behavior and test cases definition, (ii) DiaMH and test suites development, (iii) experiment execution.
- *Internal validity* threat. To limit this threat, a systematic approach was adopted to define the test scenarios, to generate realistic input data, and to apply the mutation technique.
- *External validity* threat. To limit this threat, DiaMH was developed by following the descriptions of existing diabetes control systems [PAR16, IHPS11], also providing functionalities close to IoT systems used in other domains (in general GUI-equipped systems composed of sensors, actuators and control logics).

8.5 EXPERIMENTAL COMPARISON

The analysis of the testing approach outlined in this chapter was extended by comparing its effectiveness against a runtime verification approach for IoT systems that was partially developed by some of the authors that contributed to the testing approach [LAF+18].

Both proposals require, as first step, to specify the expected behavior of the IoT system under validation/verification by means of a UML state machine, which plays a key role for the subsequent steps.

The set of generated mutants discussed in Section 8.4 was augmented, by considering also further unexpected behaviors in the Node-RED flows implementing the healthcare cloud system, in the Java app running on the smartphones, and in the network communication among the simulated devices.

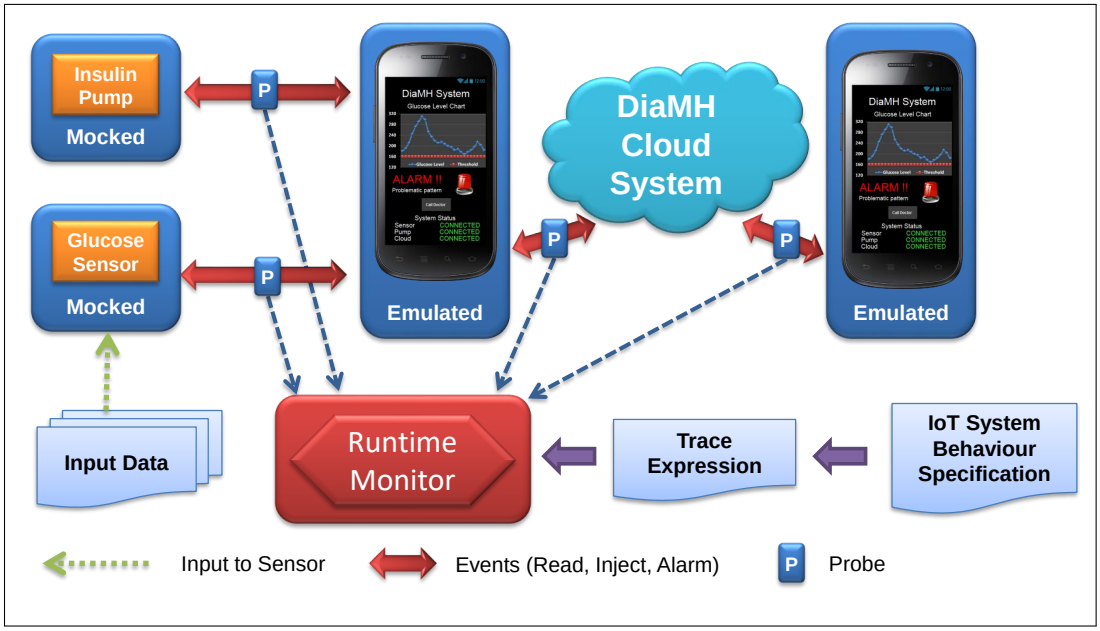
The evaluation aimed at comparing the effectiveness of the two approaches for better understanding their strengths and weaknesses. For that purpose, the testing framework was limited to Appium, since the results by applying respectively Sikuli and Appium were very similar (see Table 8.1).

In the following, the runtime verification approach is briefly described (please refer to [LAF+18] for a more complete explanation), as well as the research question, the procedure, the results, and the threats to validity of the comparison.

8.5.1 Runtime Verification Approach

Figure 8.10 reports an overview of the elements involved in the runtime verification approach for DiaMH; differently from Figure 8.2, the testware is replaced by a *trace expression*, a formalism used in runtime verification approaches to express the interactions occurring in a system during its execution and to verify its properties at runtime [AFM16], and by a *runtime monitor* observing all the relevant events by means of *probes* (P).

Figure 8.10: Elements involved in the Runtime Verification Approach.



8.5.1.1 Trace Expression

Starting from the UML state machine of Figure 8.3, the runtime verification approach requires to define an equivalent trace expression. This task can be manually performed by an expert knowing

Figure 8.11: Trace expression for DiaMH.

$$
\begin{aligned}
&Main = Normal\langle[]\rangle \\
&Normal\langle L\rangle = \{let\ L';\ \text{if } read(L, L', 4) \text{ then } inject : Discard_5 \cdot More\langle[]\rangle \\
&\qquad\qquad \text{else } read(L, L', 0) : Normal\langle L'\rangle\} \\
&More\langle L\rangle = \{let\ L';\ \text{if } read20(L, L', 16) \text{ then } alarm : inject : Discard_5 \cdot Problem\langle[]\rangle \\
&\qquad\qquad \text{else if } read20(L, L', 4) \text{ then } inject : Discard_5 \cdot More\langle[]\rangle \\
&\qquad\qquad \text{else if } read20(L, L', 0) \text{ then } Normal\langle L'\rangle \\
&\qquad\qquad \text{else } read(L, L', 0) : More\langle L'\rangle\} \\
&Problem\langle L\rangle = \{let\ L';\ \text{if } read20(L, L', 16) \text{ then } alarm : inject : Discard_5 \cdot Problem\langle[]\rangle \\
&\qquad\qquad\quad \text{else if } read20(L, L', 4) \text{ then } inject : Discard_5 \cdot More\langle[]\rangle \\
&\qquad\qquad\quad \text{else if } read20(L, L', 0) \text{ then } Normal\langle L'\rangle \\
&\qquad\qquad\quad \text{else } read(L, L', 0) : Problem\langle L'\rangle\} \\
&Discard_{i>0} = ignore : Discard_{i-1} \qquad\qquad Discard_0 = \epsilon
\end{aligned}
$$

both the UML state machine and the trace expression formalisms. Figure 8.11 reports the trace expression for DiaMH, formalizing the UML state machine.

For instance, the event $read(L, L', n)$ matches an input from the glucose sensor with the following constraints: L represents the list of already received values, L' is the previous list with the addition of the new value (notice that, to be compliant with the state machine of Figure 8.3, if more than 20 values are in the list, the oldest one is removed), and finally n indicates the minimum number of glucose values that must be over the threshold to evaluate $read$ as true. The $inject$ event represents an insulin injection, which is always followed by a discard action of the next 5 readings ($Discard_5$); the $alarm$ event instead occurs in case a problematic pattern is detected. The $read20(L, L', n)$ event expects the resulting L' list to be of length 20. Finally, [] denotes the empty list [LAF+18]. The trace expression shown in Figure 8.11 was implemented using Prolog by some of the authors that contributed to the testing approach [LAF+18].

A portion of the trace expression implemented in Prolog is shown in Listing 8.1. As shown, the **`trace_expression`** predicate handles the states of the system, which starts from the Normal state with an empty array of glucose values. In case 4 received values are greater or equal the threshold (**`read(var(l), var(l1), 4)`**), the simulated pump event arises (**`pump`**), followed by the discard of the next 5 values received from the sensor (**`ignore`** events within **`Discard`** label); then, the system moves to the More Insulin state with a new empty array. In the other case, the system stays in the Normal state, and the array is updated with the newly received value.

Listing 8.1: A portion of trace expression in Prolog.

```
...
trace_expression(Main) :-
  Main = app(Normal, []),
  Normal = gen(l, var(l1, ifelse(read(var(l), var(l1), 4),
    (pump : Discard) * app(More, []),
    read(var(l), var(l1), 0) : app(Normal, var(l1))
  ))),
  More = ...,
  Problem = ...,
  Discard = ignore : ignore : ignore : ignore : ignore : eps.
```

8.5.1.2 Runtime Monitor

The runtime monitor observes all the relevant events occurred during the system execution and matches the resulting trace against the trace expression.

The monitor was implemented as an HTTP server able to serve two kinds of requests: *check-event* and *reset*. A *check-event* is a POST request including a JSON object to describe the currently intercepted event (e.g., {"event":"sensor", "value":v}), which must be checked by the monitor to determine if the event is legal w.r.t. the trace expression, while a *reset* is a GET request used to reset the monitor state. On the client side, a *probe node* was implemented in Node-RED to represent the probes; the probe node takes in input a JavaScript object representing an event, stringifies it in JSON, sends a *check-event* request to the monitor with the corresponding JSON object, and outputs the JSON object returned by the monitoring server as response, which tells whether the observed event matches the specification.

For the monitor implementation SWI-Prolog[16] was used, which is a natural choice to support the formalism of trace expressions, since the rules of the labeled transition system defining the operational semantics of trace expressions can be directly expressed in Prolog.

8.5.2 Research Question

The comparison aims at answering the following research question **RQ1**.

RQ1: *What is the capability of the acceptance testing and runtime verification approaches in detecting bugs in an IoT system?*

The goal is to quantify and compare the capability of the two approaches in revealing bugs in IoT systems; moreover, in this way, it should be possible to evaluate the differences (if any)

[16] http://www.swi-prolog.org

between them. This would give developers and project managers an idea of the bug-detection capability achievable by following the two approaches. The technique adopted for evaluating the effectiveness is, again, mutation testing, while the metric used to answer this research question is the percentage of killed mutants out of the total. Since the location of a bug in an IoT system may affect the capability of the two approaches in revealing it, and given that IoT systems heavily rely on the network, the research question **RQ1** can be structured into three sub-research questions:

What is the capability of the acceptance testing and runtime verification approaches in:

- **RQ1.1** *detecting bugs in the cloud portion of an IoT system (see DiaMH Cloud System in Figures 8.2-8.10)?*
- **RQ1.2** *detecting bugs in the app running on the mobile devices composing an IoT system (see Emulated Smartphones in Figures 8.2-8.10)?*
- **RQ1.3** *detecting network problems among the devices composing an IoT system (see Red Arrows Events in Figures 8.2-8.10)?*

8.5.3 Procedure

Starting from the implementation of DiaMH and the two kinds of SQA artifacts created by following the approaches (i.e., the test scripts and the runtime monitor), the procedure is the following.

To answer **RQ1.1**, Stryker mutation tool was used again, generating 56 mutants in total. More specifically:

- *Mutating Javascript functions nodes*: for this case, the same mutants generated for the evaluation of the acceptance testing approach were used, resulting in 29 mutated versions of DiaMH, hence excluding the 10 equivalent ones from the originally generated 39 mutants (see the Procedure in 8.4.4 and Table 8.1).
- *Mutating switch nodes*: the logic embedded in the switch nodes used in the original Node-RED flows was translated as if-then-else Javascript statements. Such statements were mutated with Stryker, obtaining 27 mutants and, consequently, 27 corresponding mutated versions of DiaMH.

To answer **RQ1.2**, MDroid+[17] [LVBT+17] mutation tool was selected to mutate the Java code implementing the DiaMH app running on the smartphones, with assignments substitutions, logical operators changes and conditionals removals, resulting in 40 mutants.

To answer **RQ1.3**, 7 versions of DiaMH were manually created to simulate network problems, i.e., delayed and undelivered messages. More in detail: 2 versions simulating delays by 2 and 4 seconds of each network message sent from the glucose sensor to the cloud, 2 versions simulating

[17] https://research-appendix.com/mdroid

delays by 2 and 4 seconds of each network message sent from the cloud to the insulin pump, and finally 3 versions simulating an undelivered message every 5, 10, and 15 messages sent from the glucose sensor to the cloud.

Finally, each mutated version of DiaMH (103 overall), was validated separately by both the SQA artifacts produced with the two approaches, i.e., the Appium test scripts and the Prolog monitor, using the proper input data to instrument the mocked glucose sensor, in order to purposely exercise exactly one by one the 10 interesting paths of DiaMH, see 8.3.3, resulting in 1030 overall executions (10 interesting paths for 103 mutants) for each approach, noting down: (i) whether the mutant was killed, and if so, during the execution of which interesting path, and (ii) the results of a detailed analysis to explain why each mutant was killed or not.

8.5.4 Results

Table 8.3 column **RQ1.1 Cloud Mutants Killed** summarizes the number of mutants in the cloud killed by the Appium test scripts and by the Prolog monitor. Among the generated 56 mutants, 14 outlived acceptance testing and 12 outlived runtime verification.

Table 8.3: Mutants killed by the Appium test scripts and by the Prolog monitor.

Interesting Paths		# Transitions	RQ1.1: Cloud Mutants Killed		RQ1.2: App Mutants Killed		RQ 1.3: Network Mutants Killed		Total Mutants Killed	
			Acceptance Testing	Runtime Verification	Acceptance Testing	Runtime Verification	Acceptance Testing	Runtime Verification	Acceptance Testing	Runtime Verification
from starting the app (S) to Normal		1	4	1	12	0	0	0	16	1
from S to More Insulin		2	10	19	14	7	0	4	24	30
from S to Problematic		3	32	41	20	7	2	7	54	55
from S to More Insulin and back to Normal		3	19	39	14	7	1	6	34	52
from S to Problematic and directly to Normal		4	41	43	20	7	2	7	63	57
from S to Problematic and back to More Insulin		4	37	43	20	7	1	7	58	57
from S to Problematic and back to Normal (via More Insulin)		5	42	44	21	7	4	7	67	58
from S to self-loop to Normal		2	4	4	12	0	0	0	16	4
from S to self-loop to More Insulin		3	19	38	14	7	1	7	34	52
from S to self-loop to Problematic		4	34	42	18	7	4	7	56	56
Total Mutants killed	**(a)**		**42**	**44**	**25**	**7**	**5**	**7**	**72**	**58**
Total number of Mutants			**56**	**56**	**40**	**40**	**7**	**7**	**103**	**103**
Total number of Mutants (excluding equivalent)	**(b)**		**48**	**48**	**25**	**25**	**7**	**7**	**80**	**80**
Mutants detection rate	**(a/b)**		**88%**	**92%**	**100%**	**28%**	**71%**	**100%**		

As in the analysis conducted in 8.4.5, results were influenced by equivalent mutants and weakness in input data.

Each outliving mutant was analyzed and, after some code inspection, it was discovered that the behavior of 8 of them was exactly equivalent to the original system [GSZ09, JH11], hence undetectable by any black-box SQA approach. For instance, there was a mutant changing if (i==20) i=0 to if (i≥20) i=0 in a Node-RED function node; since the condition is evaluated for each single increment of i, and the initial value of i is below 20, the behavior of the mutant is equivalent to the original code.

Thus, only 6 and 4 mutants were considered as real survivors for acceptance testing and runtime verification approaches, respectively. From the analysis, it was discovered that 3 of them survived,

in both the approaches, due to weaknesses in the provided input data, which was not complete enough to cover all the possible conditions and properly exercise the boundaries of the original system (see 8.4.5 where the results were affected in a similar way). Indeed, mutations often affect operators used in conditions; if the mutation drastically changes the system behavior (e.g., $>$ in $<$), the mutant can be easily detected, but if it is just a little variation of the system behavior (e.g., $>$ in $\geq$), the input data must be carefully chosen in order to detect the inconsistency. To test this conjecture, an ad-hoc sequence of values was created for each of the 3 mutants that survived because of weak input data, and all of them were identified by both the approaches. Concerning the 3 other mutants surviving only the acceptance testing approach, they were identified as slight mutations of the DiaMH behavior having no effect on the GUI, hence impossible to be killed by the test scripts, that are highly based on assertions over GUI properties. For example, a mutant changed the number of readings required to determine a patient's state from 20 to 19: while no test script has assertions to verify the exact number of received readings that brings to a change of a patient's state, the Prolog monitor looks at each message exchanged among the devices and can identify when something expected is missing. Finally, the last mutant surviving only runtime verification had caused the crash of the app and the freezing of the GUI at the end of its execution, due to a continuous and erroneous incrementation of the size of an array. In that case, while the mutant was detected by some test scripts, since the GUI had stopped displaying the proper content after the crash, no unexpected events were detected by the monitor.

Table 8.3 column **RQ1.2 App Mutants Killed** summarizes the number of mutants in the app running on the smartphones killed by the Appium test scripts and by the Prolog monitor. Among the generated 40 mutants, 15 and 33 outlived acceptance testing and runtime verification, respectively. As for **RQ1.1**, they were analyzed, finding that 15 out of 40 were equivalent. For example, there were mutants applying trivial and undetectable GUI changes (e.g., font color/style or unchecked visual content) or initializing variables by negative or null values, which were reinitialized before their usages. Hence, excluding the equivalent mutants, in this case, the acceptance testing approach resulted by far more effective than the runtime verification (100% versus 28%), because the 18 mutants out of 33 surviving runtime verification involved changes in the GUI of the app without affecting the components communication (e.g., the label describing the current patient's state was forcefully changed by a mutant independently of the exchanged glucose values). These mutants could not be detected by the Prolog monitor, which was able instead to detect those producing unexpected events altering the system expected behavior, as in the case of a mutant setting to false a conditional expression used for establishing the communication.

Table 8.3 column **RQ1.3 Network Mutants Killed** summarizes the number of mutants concerning network problems, i.e., delayed and undelivered messages, killed by the Appium test scripts and by the Prolog monitor. In this case, the runtime verification approach was able to detect all the mutants, while the acceptance testing approach detected only 5 out of 7. The surviving mutants were those respectively delaying by 2 seconds the messages sent from the glucose sensor to the cloud and from the cloud to the insulin pump. Differently from the runtime verification approach, the acceptance testing approach required to introduce some temporal waits into the

test scripts (e.g., to perform a refresh of the GUI), which affected their actual execution time and reduced the overall precision in detecting small timing deviations from the DiaMH expected behavior. Concerning undelivered messages, for the acceptance testing approach, only the test script exercising the longest interesting path (i.e., from S to Problematic and back to Normal (via More Insulin)) was able to kill the mutant which simulated an undelivered message every 15 messages sent from the glucose sensor to the cloud. Instead, the SQA artifacts exercising shorter interesting paths (e.g., from starting the app (S) to Normal and from S to self-loop to Normal) were not able to kill any mutant concerning network problems, neither in acceptance testing nor in runtime verification, since they used input data having very similar values and their execution ended before the mutation had occurred.

Summary: To summarize, the results show the effectiveness of both acceptance testing and runtime verification approaches. Acceptance testing proves to be by far more effective than runtime verification in detecting mutants in the app affecting the GUI (100% versus 28% for acceptance testing, **RQ1.2**). Instead, runtime verification is much more precise in tracking subtle changes in the system behavior, in particular in the messages exchanged among the DiaMH components: indeed, it is slightly better in killing mutants in the cloud (92% versus 88% for runtime verification, **RQ1.1**) and those concerning network problems, where even small deviations from the expected behavior, like delayed and undelivered messages, are detected (100% versus 71% for runtime verification, **RQ1.3**). If the equivalent outliving mutants are excluded (8 from the cloud and 15 from the app, see Table 8.3) from the total (56 from the cloud, 40 from the app, and 7 from the network, see Table 8.3), by combining the approaches, only 3 mutants out of 80 survive (see results of **RQ1.1** concerning input data selection), hence over 96% of the generated mutants that actually modify the behavior of DiaMH are killed.

The results hint that each approach can be chosen depending on the kind of IoT system that has to be developed and validated/verified. In case a strong interaction between the user and the system is expected, or when the system includes a GUI displaying several messages and data, the acceptance testing approach is preferable. Instead, the runtime verification approach is more effective when a precision in the timing during a communication among the devices is required or when a certain order of events is expected. In many cases, both approaches may be combined to improve the overall capability in detecting bugs in IoT systems.

8.5.5 Threats to Validity

The threats of the present study are basically the same listed in 8.4.6. In particular, the *Authors' bias* threat was limited by dividing among the authors of the two approaches the various tasks to apply on the case study, as well as by employing existing automated mutation tools (Stryker and MDroid+) with default settings.

Chapter 9

Related Work

As discussed in the previous chapters, no works expressively address quality assurance of IoT systems in Node-RED. Nevertheless, the guidelines presented in Chapter 6 have been inspired by several proposals originating from different contexts, and using different notations, for example UML and BPMN to improve the quality of produced models. The guidance provided to Node-RED developers in implementing nodes is indeed limited to a simple set of principles[1], like nodes should be "simple to use" and "consistent" in their behavior, and few unofficial design patterns[2] to make flows easier to understand and reuse.

In a recent industrial work [BCAP19], Bröring *et al.* propose an approach to automatically collect meta-data from Node-RED flows and nodes, and feed a knowledge base for future analysis, such as nodes quality ratings, downloads data, and nodes dependencies. Although this work considers several quality aspects of Node-RED, like selecting the most suitable solutions to integrate within a system, it does not provide to users any development guideline.

Prehofer and Chiarabini [PC15] identify the differences between mash-up tools for IoT systems, like Node-RED, and model-based approaches for the IoT, and propose an approach to exploit both their benefits: the simplicity of mash-up tools in systems development and the strengths of models to formalize a behavior and have it checked by a model checker. However, in this paper, the quality checks of IoT systems are not oriented to the comprehensibility issues that may emerge during flows development using mash-up tools.

Mendling *et al.* [MRvdA10] provide 7 guidelines, built on empirical insights, as a response to the lack of practical solutions to improve the quality of business process models. Some of these guidelines have been adapted to formulate the guidelines discussed in Chapter 6. For example, "Use verb-object activity label" (G6), to reduce the ambiguity of the constructs in a model, is particularly useful for the large number of nodes collaborating within Node-RED flows, as well

[1] `https://nodered.org/docs/creating-nodes`

[2] `https://medium.com/node-red/node-red-design-patterns-893331422f42`

as “Use as few elements in the model as possible” (G1) and “Decompose a model with more than 50 elements” (G7), helpful to reduce flows complexity.

Unhelkar, in his book [Unh05], focuses on syntax, semantics and aesthetic checks of UML 2.0 diagrams. Although UML is quite different from Node-RED in many aspects, since it operates at a design stage and involves constructs that are hardly comparable to Node-RED nodes and wires, in the book there are some aesthetic checks concerning activity diagrams that have been used to formulate the guidelines of Chapter 6. In particular, it is important to adopt a consistent style to differentiate regular from exceptional scenarios and to balance overpopulated diagrams by redistributing the included constructs.

Reggio *et al.* [RLRA12, RLR11] face the problem of quality in business process modeling. They propose an empirical method for helping the modeler in choosing among five business process modeling styles, that differ in terms of abstraction and precision. For instance, a more precise style requires each construct to declare all the participants, the objects and the used data in capital letters, to make explicit the data used by each node. Some of these principles have been adapted to the work discussed in Chapter 6.

Ambler proposes [Amb05] several guidelines addressing both general and UML-specific modeling issues, with the aim of improving the effectiveness of the produced models. Some of these guidelines can be applied even in Node-RED, since they use very general terms and descriptions. Limiting to UML activity diagrams, since they represent sequences of actions similarly to Node-RED flows, Ambler’s guidelines suggest, for instance, to avoid black-hole and miracle nodes (i.e., nodes without a leaving/entering line), that may indicate a missing interaction, and to check that the guards within decision points are always complete.

Concerning IoT system testing, very few proposals treat such activity from a high level perspective, like acceptance testing, and even if some frameworks based on the Node.js environment have been recently developed, no systematic approaches specifically address Node-RED either. More importantly, no proposal tries to exploit the simplicity and the expressive power of Node-RED, that acts very close to the design stage of a system, to guide implementation and testing activities.

A graphical tool is implemented by Hoxha *et al.* [HBA+14] to guide the user in formulating, visualizing, and monitoring temporal logic sentences, each one expressing design requirements properties of a Cyber-Physical System (e.g., reachability, safety). The work clearly differs from the approaches presented in Chapters 7 and 8 in terms of goal and focus, which here is to present the implemented tool. Moreover, the design requirements properties are based on temporal logic sentences.

Kim *et. al* [KAH+18] introduce a service-based framework for testing IoT systems, by adapting and evolving traditional testing methodologies to the context of IoT. The goals of the paper are different from the proposed approaches, and do not directly answer problems concerning IoT systems development, in particular in Node-RED.

A Parasoft white paper [PAR16] formulates the idea of de-constructing an healthcare system into layers for testing purposes, isolate the various components to improve automation and use a service testing solution for testing these components. The testing strategy is based on stimulating the wired sensors to propagate to the system the obtained values and on verifying if any alert emerges. In this idea, any interaction between the components is hidden, while both proposals outlined in Chapters 7 and 8 require a formalization of the system behavior, including any interesting interaction occurring among the components, to generate compliant test cases.

Kane *et al.* [KFK14] present a runtime verification technique to describe and detect properties violations of safety-critical systems, formally described. Their research challenges are very different from those addressed in Chapters 7 and 8; they concern, for instance, how to properly abstract a system based on a limited perspective of its internal behavior and how such abstraction is close enough to the real system.

Rosenkranz *et al.* in [RWBO15] present a test system architecture for open-source IoT software, sketching some challenges, like heterogeneity of hardware and interoperability testing. In this work, they recognize the two testing levels in IoT system testing, i.e., virtualization of devices and testing the system under real conditions, but a systematic approach for testing IoT system from unit or acceptance level is not present.

A model-based testing architecture is proposed by Silva *et al.* [SPB+14] for generating regression models to simulate patients vital signs and emulate medical devices and actuators, by means of stored clinical data, medical guidelines and statistical techniques. A controller model is introduced to check the events and coordinate the actions among the devices and the patient models, which can also be reused and adapted depending on the clinical scenario (e.g., a patient with a completely different pathology). In this work, patients and devices are described by models, and the patient's condition is determined by the change of states and described in terms of patterns. While in the approaches presented in Chapters 7 and 8 the goal is to obtain executable test cases aligned with the specifications, either from a unit or an acceptance testing level, the aim of this paper is simulating patients vital signs.

Simulation modeling aimed at abstracting and testing complex systems composed of interconnected devices is indeed a widespread solution [ASEZ17, AIH15], since it can be adapted in many contexts and combined with other approaches. Test oracles and test data can be generated from some predefined rules and injected into sensors and actuators to replicate safety-critical scenarios [AWM+17]. Arrieta *et al.* [ASEZ17] present a methodology and a tool to automatically generate and instantiate test systems based on simulation models by introducing test oracles and test data, for example to test Cyber-Physical Systems in terms of concurrency and timing responses properties. In a next work [AWM+17], they face the unpredictability of Cyber-Physical Systems interactions and the cost of generating a comprehensive simulation model, presenting a search-based approach built on top of a multi-objective genetic algorithm, to generate test cases guided by three cost-effectiveness measures: requirements coverage, test execution time and test case similarity. Some mutation operators are also introduced to test variations of the generated

stimulus. Even though, in particular in the last work, they adopt a mutation mechanism to exercise the test artifacts, and they consider a requirements coverage criteria, the produced testing artifacts are not GUI-based, and the general purposes are neither about acceptance testing nor try to address alignment between requirements and test cases.

Novák *et al.* [NKW17] adopt the concept of agents and multi-agents systems (MASs) to simulate complex hybrid Cyber-Physical production environments and overcome the heterogeneity problem while simulating such systems. Individual agents are used to represent singular components at discrete times, while their integration is modeled through a MAS. The approach is promising as an alternative to mock system components, but in this case there is more concern about simulation of interconnected devices than actual testing artifacts generation. Moreover, the paper does not explicitly include a method for generating test cases aligned with a system specifications.

Giménez *et al.* [GMPE13] developed a Java sensor simulator to test the feasibility of under deployment data-centric applications based on sensors and to assess the safety of a simulated industrial environment. Multiple kinds of sensors (e.g., mobile, temperature and humidity) are configured via XML files, to reproduce interesting scenarios before an actual environment deployment, and are fed by data generated from a set of patterns. Sensors can react depending on some given patterns and can be specialized depending on the scenario that has to be reproduced, but the paper is more oriented to safety and deployment feasibility than actual generation of test cases.

Siboni *et al.* [SST+16] propose and evaluate a security testbed framework for testing wearable devices in terms of security design requirements, where external attackers and sensors are simulated and stimuli are generated accordingly to the testing purposes. Differently from the proposals in Chapters 7 and 8, the focus of this work is security testing.

Chapter 10

Conclusion and Future Work

The preliminary set of guidelines introduced in Chapter 6 addresses some common Node-RED comprehensibility issues. They may help the Node-RED developers in producing flows that are easier to comprehend and suitable for future maintenance and testing activities. The guidelines have been evaluated with an experiment that involved ten master students. The results have shown that the adoption of the guidelines increases the overall efficiency in Node-RED flows comprehension, by reducing the number of errors and the time required to complete comprehension tasks over some provided Node-RED flows. In addition to supporting the Node-RED developers in flows development, the proposed guidelines pinpoint some Node-RED comprehensibility issues that might be solved in future releases of the tool, by adding additional features, such as nodes resizing in height and width, and graphical jumps between wires. The proposed guidelines are just a preliminary set of those that will be formulated to address further Node-RED comprehensibility issues that did not emerge from the experiment in Chapter 6; for instance, new guidelines may propose an alternative design to avoid loops by transforming graph-based flows into tree-based ones. A larger participants pool will be involved in future experimental evaluations, including Node-RED designers. Finally, a checker tool will be implemented to automatically detect the comprehensibility issues from the Node-RED flows failing the proposed guidelines, and fix them accordingly.

In Chapter 7 a preliminary version of an approach for developing and testing IoT systems in Node-RED has been proposed. First, the static and dynamic aspects of the system are modeled via UML class and activity diagrams, to describe the properties of the nodes and the flows structures. Then, from the model, it is possible to generate the executable Node-RED flows implementing the system and to perform an iterative testing activity aimed at: (1) selecting a set of test scenarios from the model, (2) defining some control points within the selected test scenarios to check over the system properties, and (3) generating the corresponding Javascript test scripts to exercise the selected test scenarios in the Mocha test framework. The approach will be fine-tuned by investigating the subtle differences between UML and Node-RED and will integrate some of the

guidelines proposed in Chapter 6 to support the production of understandable and of high quality models. Moreover, the testing activity will be improved with a strategy for selecting effective test scenarios, for generating smart input data tailored for the test scenarios, and for formulating more complex assertions. Finally, the tasks of the approach that are intended to be automated (i.e., the generations of the Node-RED flows and the test scripts) will be implemented in a prototype tool.

Chapter 8 has presented an approach for acceptance testing of GUI-equipped IoT systems, and its application and evaluation on a realistic IoT case study monitoring diabetic patients, composed of local sensors and actuators, a cloud-based healthcare system and an Android application. The approach is applicable to all GUI-equipped IoT systems that rely on a GUI as a principal way of interaction between the user and the system. The starting point is the formalization of the system expected behavior, by means of a UML state machine, which drives the development/virtualization of the system (employing, e.g., Node-RED), and the generation of the test artifacts from its paths. The effectiveness of the test suites produced using two different testing frameworks has been empirically evaluated in terms of bugs detection by using mutation testing. Results have shown the effectiveness of the approach in killing 93% of the generated mutants affecting a specific portion of the IoT system (i.e., the virtualized cloud system). The approach has been compared against a runtime verification approach, to determine its strengths and weaknesses: again, the acceptance testing approach has resulted effective in detecting different kind of bugs, from network communication issues to unexpected changes in the GUI, ranging between 71% and 100% of mutants killed. As future research, the approach will be applied on other IoT systems to verify its applicability and scalability.

Part IV

Bibliography

Bibliography

[ABBB15] Roberto Acerbis, Aldo Bongio, Marco Brambilla, and Stefano Butti. Model-driven development based on OMG's IFML with WebRatio web and mobile platform. In Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe, editors, *Proceedings of 15th International Conference on Web Engineering (ICWE 2015)*, volume 9114 of *LNCS*, pages 605–608. Springer, 2015.

[ACRR07] Egidio Astesiano, Maura Cerioli, Gianna Reggio, and Filippo Ricca. A phased highly-interactive approach to teaching UML-based software development. In *Proceedings of Educators Symposium at MoDELS 2007*, pages 9–18. University of Goteborg, 2007.

[AFM16] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. Comparing Trace Expressions and Linear Temporal Logic for runtime verification. In *Theory and Practice of Formal Methods*, pages 47–64. Springer, 2016.

[AIH15] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for Cyber Physical System. In *IFIP International Conference on Testing Software and Systems*, pages 194–207. Springer, 2015.

[Ale04] Alvin Alexander. ACME Original Requirements Specification, version 6.1. `http://alvinalexander.com/java/misc/ReEnableExternalUser/`, 2004.

[Amb05] Scott W Ambler. *The elements of UML (TM) 2.0 style*. Cambridge University Press, 2005.

[AR02] E. Astesiano and G. Reggio. Knowledge structuring and representation in requirement specification. In *Proceedings of 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE 2002, pages 143–150. ACM, 2002.

[ASEZ17] Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria, and Justyna Zander. Automatic generation of test system instances for configurable Cyber-Physical systems. *Software Quality Journal*, 25(3):1041–1083, 2017.

[AWM+17] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Search-based test case generation for Cyber-Physical systems. In *Evolutionary Computation (CEC), 2017 IEEE Congress on*, 2017.

[BBC10] Felipe M. Besson, Delano M. Beder, and Marcos L. Chaim. An automated approach for acceptance web test case modeling and executing. In *Proceedings of 11th International Conference on Agile Software Development (XP 2010)*, volume 48 of *LNBIP*, pages 160–165. Springer, 2010.

[BCA18] Miroslav Bures, Tomás Cerný, and Bestoun S. Ahmed. Internet of Things: Current challenges in the quality assurance and testing methods. *CoRR*, abs/1805.01241, 2018.

[BCAP19] Arne Bröring, Victor Charpenay, Darko Anicic, and Sebastien Püech. Nova: A knowledge base for the Node-RED IoT ecosystem. In *The Semantic Web: ESWC 2019 Satellite Events*, pages 257–261. Springer International Publishing, 2019.

[BM83] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[CHLX09] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(1):24, 2009.

[Cle16] Diego Clerissi. Acceptance Test Driven Development Approach Applied: Address-Book Case Study. `https://sepl.dibris.unige.it/2016-ATDD.php`, 2016.

[CLR20] Diego Clerissi, Maurizio Leotta, and Filippo Ricca. A set of empirically validated development guidelines for improving Node-RED flows comprehension. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2020. (Accepted).

[CLRR] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. Phonebook textual and UML-based requirements specifications. `http://sepl.dibris.unige.it/2017-RET.php`.

[CLRR16a] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. A lightweight semi-automated acceptance test-driven development approach for web applications. In Alessandro Bozzon, Philippe Cudré-Mauroux, and Cesare Pautasso, editors, *Proceedings of 16th International Conference on Web Engineering (ICWE 2016)*, volume 9671 of *Lecture Notes in Computer Science*, pages 593–597. Springer, 2016.

[CLRR16b] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. Test driven development of web applications: A lightweight approach. In *Proceedings of QUATIC 2016*, pages 25–34. IEEE, 2016.

[CLRR17] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. Towards the generation of end-to-end web test scripts from requirements specifications. In *IEEE 25th International Requirements Engineering Conference Workshops, RE 2017 Workshops, Lisbon, Portugal, September 4-8, 2017*, pages 343–350, 2017.

[CLRR18] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. Towards an approach for developing and testing Node-RED IoT systems. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Ensemble-Based Software Engineering, EnSEmble@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 4, 2018*, pages 1–8, 2018.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2000.

[CR04] Christine Choppy and Gianna Reggio. Improving use case based requirements using formally grounded specifications. In Michel Wermelinger and Tiziana Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 244–260. Springer, 2004.

[Dow11] Gary Downs. Lean-agile acceptance test-driven development: Better software through collaboration by Ken Pugh. *ACM SIGSOFT Software Engineering Notes*, 36(4):34–34, 2011.

[FNB07] J. Ferreira, J. Noble, and R. Biddle. Agile development iterations and UI design. In *Proceedings of Agile Conference*, AGILE 2007, pages 50–58, 2007.

[GEL+16] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. Arsenal: Automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*, pages 41–46. Springer, 2016.

[GK05] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[GMPE13] Pablo Giménez, Benjamin Molina, Carlos E Palau, and Manuel Esteve. SWE simulation and testing for the IoT. In *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*, pages 356–361. IEEE, 2013.

[GOP12] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[GSZ09] Bernhard JM Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 192–199. IEEE, 2009.

[HBA+14] Bardh Hoxha, Hoang Bach, Houssam Abbas, Adel Dokhanchi, and Yoshihiro Kobayashi. Towards formal specification visualization for testing and monitoring of Cyber-Physical systems. In *In Int. Workshop on Design and Implementation of Formal Tools and Systems*, 2014.

[HHM11] Theodore D. Hellmann, Ali Hosseini-Khayat, and Frank Maurer. Test-driven development of graphical user interfaces: A pilot evaluation. In *Proceedings of 12th International Conference on Agile Software Development (XP 2011)*, pages 223–237, 2011.

[HRT16] M. Hammoudi, G. Rothermel, and P. Tonella. Why do record/replay tests of web applications break? In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2016)*, page (in press). IEEE, 2016.

[HS91] H. Rex Hartson and Eric C. Smith. Rapid prototyping in human-computer interface development. *Interacting with Computers*, 3(1):51–91, 1991.

[IHPS11] R. Istepanian, S. Hu, N. Philip, and A. Sungoor. The potential of Internet of m-health Things "m-IoT" for non-invasive glucose level sensing. In *33rd International Conference of the IEEE Engineering in Medicine and Biology Society*, EMBC 2011, pages 5264–5266, 2011.

[IKK+15] SM Riazul Islam, Daehan Kwak, MD Humaun Kabir, Mahmud Hossain, and Kyung-Sup Kwak. The Internet of Things for Health care: A comprehensive survey. *IEEE Access*, 3:678–708, 2015.

[JD11] Mingyue Jiang and Zuohua Ding. Automation of test case generation from textual use cases. In *Interaction Sciences ICIS 2011*, pages 102–107. IEEE, 2011.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[KAH+18] Hiun Kim, Abbas Ahmad, Jaeyoung Hwang, Hamza Baqa, Franck Le Gall, Miguel Angel Reina Ortega, and JaeSeung Song. Iot-taas: Towards a prospective IoT testing framework. *IEEE Access*, 6:15480–15493, 2018.

[KFK14] Aaron Kane, Thomas Fuhrman, and Philip Koopman. Monitor based oracles for Cyber-Physical system testing: Practical experience report. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and*

Networks, DSN '14, pages 148–155, Washington, DC, USA, 2014. IEEE Computer Society.

[Klo13] David C. Klonoff. The current status of mHealth for Diabetes: Will it be the next big thing? *Journal of Diabetes Science and Technology (JDST)*, 7(3):749–758, 2013.

[KR03] Supaporn Kansomkeat and Wanchai Rivepiboon. Automated-generating test case using UML statechart diagrams. In *Proceedings of SAICSIT 2003*, pages 296–300. South African Institute for Computer Scientists and Information Technologists, 2003.

[KT06] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.

[KTL15] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 560–564. IEEE, 2015.

[LAF+18] Maurizio Leotta, Davide Ancona, Luca Franceschini, Dario Olianas, Marina Ribaudo, and Filippo Ricca. Towards a runtime verification approach for Internet of Things systems. In *Proceedings of 2nd International Workshop on Engineering the Web of Things (EnWoT 2018)*, LNCS. Springer, 2018.

[LCF+19] Maurizio Leotta, Diego Clerissi, Luca Franceschini, Dario Olianas, Davide Ancona, Filippo Ricca, and Marina Ribaudo. Comparing testing and runtime verification of IoT systems: A preliminary evaluation based on a case study. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 434–441. SCITEPRESS-Science and Technology Publications, Lda, 2019.

[LCO+18] Maurizio Leotta, Diego Clerissi, Dario Olianas, Filippo Ricca, Davide Ancona, Giorgio Delzanno, Luca Franceschini, and Marina Ribaudo. An acceptance testing approach for Internet of Things systems. *IET Software*, 12(5):430–436, 2018.

[LCRS13a] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Comparing the maintainability of Selenium WebDriver test suites employing different locators: A case study. In *Proceedings of 1st International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*, JAMAICA 2013 at ISSTA 2013, pages 53–58. ACM, 2013.

[LCRS13b] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Improving test suites maintainability with the Page Object pattern: an industrial case study. In *Proceedings of 6th International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2013, pages 108–113. IEEE, 2013.

[LCRT13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. Programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013)*, pages 272–281. IEEE, 2013.

[LCRT14] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Visual vs. DOM-based web locators: An empirical study. In Marco Winckler Sven Casteleyn, Gustavo Rossi, editor, *Proceedings of 14th International Conference on Web Engineering (ICWE 2014)*, volume 8541 of *Lecture Notes in Computer Science*, pages 322–340. Springer, 2014.

[LCRT16] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Approaches and tools for automated end-to-end web testing. *Advances in Computers*, 101:193–237, 2016.

[LDCD06] Christian FJ Lange, Bart DuBois, Michel RV Chaudron, and Serge Demeyer. An experimental investigation of UML modeling conventions. In *International Conference on Model Driven Engineering Languages and Systems*, pages 27–41. Springer, 2006.

[LM95] James A. Landay and Brad A. Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, CHI 1995, pages 43–50. ACM / Addison-Wesley, 1995.

[LRC+17] Maurizio Leotta, Filippo Ricca, Diego Clerissi, Davide Ancona, Giorgio Delzanno, Marina Ribaudo, and Luca Franceschini. Towards an acceptance testing approach for Internet of Things systems. In *International Conference on Web Engineering*, pages 125–138. Springer, 2017.

[LRRA12] Maurizio Leotta, Gianna Reggio, Filippo Ricca, and Egidio Astesiano. Towards a lightweight model driven method for developing SOA systems using existing assets. In *Proceedings of 14th International Symposium on Web Systems Evolution*, WSE 2012, pages 51–60. IEEE, 2012.

[LSRT15a] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Automated generation of visual web tests from DOM-based web tests. In *Proceedings of 30th ACM/SIGAPP Symposium on Applied Computing (SAC 2015)*, pages 775–782. ACM, 2015.

[LSRT15b] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Using multi-locators to increase the robustness of web test cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, pages 1–10. IEEE, 2015.

[LSRT16] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.

[LVBT+17] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 233–244, New York, NY, USA, 2017. ACM.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[Mor10] J Paul Morrison. *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010.

[MR16] Brooke H. McAdams and Ali A. Rizvi. An overview of insulin pumps and glucose sensors for the generalist. *Journal of Clinical Medicine*, 5(1), 2016.

[MRvdA10] Jan Mendling, Hajo A Reijers, and Wil MP van der Aalst. Seven process modeling guidelines (7PMG). *Information and Software Technology*, 52(2):127–136, 2010.

[Mug08] Rick Mugridge. Managing agile project requirements with storytest-driven development. *IEEE Software*, 25(1):68–75, 2008.

[MW01] A. McDonald and R. Welland. Agile web engineering (AWE) process. Technical report, University of Glasgow, 2001.

[NKW17] Petr Novák, Petr Kadera, and Manuel Wimmer. Agent-based modeling and simulation of hybrid Cyber-Physical systems. In *3rd IEEE International Conference on Cybernetics, CYBCONF 2017, Exeter, United Kingdom, June 21-23, 2017*, pages 1–8, 2017.

[OAN14] Lukasz Olek, Bartosz Alchimowicz, and Jerzy R. Nawrocki. Acceptance testing of web applications with Test Description Language. *Computer Science (AGH)*, 15(4):459, 2014.

[O'D05] Mike O'Docherty. *Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*. Wiley, 1 edition, June 2005.

[OU01] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[PAR16] PARASOFT. End-to-end testing for IoT integrity, 2016. `https://alm.parasoft.com/end-to-end-testing-for-iot-integrity`.

[PC15] Christian Prehofer and Luca Chiarabini. From Internet of Things mashups to model-based development. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 499–504. IEEE, 2015.

[RC15] Gianna Reggio and Diego Clerissi. ACME Disciplined Requirements Specification. `http://sepl.dibris.unige.it/TR/ACME-Complete.pdf`, 2015.

[RGR+14] José Matías Rivero, Julián Grigera, Gustavo Rossi, Esteban Robles Luna, Francisco Montero Simarro, and Martin Gaedke. Mockup-driven development: Providing agile support for model-driven web engineering. *Information & Software Technology*, 56(6):670–687, 2014.

[RLR11] Gianna Reggio, Maurizio Leotta, and Filippo Ricca. "Precise is better than light" a document analysis study about quality of business process models. In *First International Workshop on Empirical Requirements Engineering, EmpiRE 2011, Trento, Italy, August 30, 2011*, pages 61–68, 2011.

[RLR14] Gianna Reggio, Maurizio Leotta, and Filippo Ricca. Who knows/uses what of the UML: A personal opinion survey. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Proceedings of 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, volume 8767 of *LNCS*, pages 149–165. Springer, 2014.

[RLR15] Gianna Reggio, Maurizio Leotta, and Filippo Ricca. A method for requirements capture and specification based on disciplined use cases and screen mockups. In *Proceedings of 16th International Conference on Product-Focused Software Process Improvement (PROFES 2015)*, volume 9459 of *Lecture Notes in Computer Science*, pages 105–113. Springer, 2015.

[RLRA12] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Egidio Astesiano. Business process modelling: Five styles and a method to choose the most suitable one. In *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling, EESSMod@MoDELS 2012, Innsbruck, Austria, October 1-5, 2012*, pages 8:1–8:6, 2012.

[RLRC14] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Diego Clerissi. What are the used activity diagram constructs? - A survey. In Luís Ferreira Pires, Slimane Hammoudi, Joaquim Filipe, and Rui César das Neves, editors, *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pages 87–98. SciTePress, 2014.

[RLRC15] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Diego Clerissi. What are the used UML diagram constructs? A document and tool analysis study covering

activity and use case diagrams. In Slimane Hammoudi, Ferreira Luís Pires, Joaquim Filipe, and César Rui das Neves, editors, *Model-Driven Engineering and Software Development*, volume 506 of *Communications in Computer and Information Science*, pages 66–83. Springer, 2015.

[RLRC18] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Diego Clerissi. DUSM: A method for requirements specification and refinement based on disciplined use cases and screen mockups. *Journal of Computer Science and Technology*, 33(5):918–939, 2018.

[RRL14] Gianna Reggio, Filippo Ricca, and Maurizio Leotta. Improving the quality and the comprehension of requirements: Disciplined use cases and mockups. In *Proceedings of 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2014)*, pages 262–266. IEEE, 2014.

[RST+10] Filippo Ricca, Giuseppe Scanniello, Marco Torchiano, Gianna Reggio, and Egidio Astesiano. On the effort of augmenting use cases with screen mockups: results from a preliminary empirical study. In *Proceedings of 4th International Symposium on Empirical Software Engineering and Measurement*, ESEM 2010, pages 40:1–40:4. ACM, 2010.

[RST+14] Filippo Ricca, Giuseppe Scanniello, Marco Torchiano, Gianna Reggio, and Egidio Astesiano. Assessing the effect of screen mockups on the comprehension of functional requirements. *ACM Transactions on Software Engineering and Methodology*, 24(1):1–38, 2014.

[RWBO15] Philipp Rosenkranz, Matthias Wählisch, Emmanuel Baccelli, and Ludwig Ortmann. A distributed test system architecture for open-source IoT software. In *Proceedings of 1st Workshop on IoT Challenges in Mobile and Industrial Systems*, IoT-Sys 2015, pages 43–48. ACM, 2015.

[SFSC16] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz Cortes. A survey on metamorphic testing. *IEEE Trans. Software Eng.*, 42(9):805–824, 2016.

[SLRT16] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. Clustering-aided Page Object generation for web testing. In Alessandro Bozzon, Philippe Cudré-Mauroux, and Cesare Pautasso, editors, *Proceedings of 16th International Conference on Web Engineering (ICWE 2016)*, volume 9671 of *Lecture Notes in Computer Science*, pages 132–151. Springer, 2016.

[SM16] Karthik Pattabiraman Shabnam Mirshokraie, Ali Mesbah. Atrina: Inferring unit oracles from GUI test cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2016)*, page (in press). IEEE, 2016.

[SMB08] Philip Samuel, R Mall, and Ajay Kumar Bothra. Automatic test case generation using UML state diagrams. *IET software*, 2(2):79–93, 2008.

[SPB+14] Lenardo C Silva, Mirko Perkusich, Frederico M Bublitz, Hyggo O Almeida, and Angelo Perkusich. A model-based architecture for testing medical Cyber-Physical systems. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 25–30. ACM, 2014.

[SRT+13] G. Scanniello, F. Ricca, M. Torchiano, C. Gravino, and G. Reggio. Estimating the effort to develop screen mockups. In *Proceedings of 39th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA 2013, pages 341–348, 2013.

[SS12] Shai Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends® in Machine Learning*, 4(2):107–194, 2012.

[SST+16] Shachar Siboni, Asaf Shabtai, Nils O Tippenhauer, Jemin Lee, and Yuval Elovici. Advanced security testbed framework for wearable IoT devices. *ACM Trans. Internet Tech. (TOIT)*, 16(4):26, 2016.

[SW65] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[SW11] Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387. IEEE, 2011.

[SZF15] M. Spichkova, A. Zamansky, and E. Farchi. Towards a human-centred approach in modelling and testing of Cyber-Physical systems. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 847–851, Dec 2015.

[TRL14] Alessandro Tiso, Gianna Reggio, and Maurizio Leotta. Unit testing of Model to Text transformations. In *Proceedings of MoDELS 2014*, pages 14–23, 2014.

[UL10] Mark Utting and Bruno Legeard. *Practical model-based testing: A tools approach*. Morgan Kaufmann, 2010.

[Unh05] Bhuvan Unhelkar. *Verification and validation for quality of UML 2.0 models*, volume 2. Wiley Online Library, 2005.

[VSBCR02] Rini Van Solingen, Vic Basili, Gianluigi Caldiera, and H Dieter Rombach. Goal Question Metric (GQM) approach. *Encyclopedia of software engineering*, 2002.

[WHO13] World Health Organization WHO. The top 10 causes of death, 2013. https://www.who.int/news-room/fact-sheets/detail/the-top-10-causes-of-death.

[WHO16] World Health Organization WHO. Global report on diabetes (1st edition), 2016. http://www.who.int/diabetes/global-report/en/.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[YAB11] Tao Yue, Shaukat Ali, and Lionel Briand. Automated transition from use cases to UML state machines to support state-based testing. In *Proceedings of ECMFA 2011*, pages 115–131. Springer, 2011.

[ZCC03] Jia Zhang, C.K. Chang, and J. Y Chung. Mockup-driven fast-prototyping methodology for web requirements engineering. In *Proceedings of 27th International Computer Software and Applications Conference*, COMPSAC 2003, pages 263–268. IEEE, 2003.

[ZE+11] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class Hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

Part V

Appendix A

Appendix A

DUSM: A Method for Requirements Specification and Refinement based on Disciplined Use Cases and Screen Mockups

This Appendix presents DUSM (Disciplined Use Cases with Screen Mockups), a method for describing and refining requirements specifications based on disciplined use cases and screen mockups. Disciplined use cases are characterized by a quite stringent template to prevent common mistakes, and to increase the quality of the specifications. Use cases descriptions are formulated in a structured natural language, which allows to reach a good level of precision, avoiding the need for further notations and complex models. Screen mockups are precisely associated with the steps of the use cases scenarios and they present the corresponding GUI (graphical user interface) as seen by the human actors before/after the steps executions, improving the comprehension and the expression of the non-functional requirements on the user interface. DUSM has been proposed and fine-tuned during several editions of a software engineering course at the University of Genova. By means of a series of case studies and experiments, the method has been validated and evaluated in terms of: (1) its effectiveness in improving the comprehension and, in general, the quality of the produced requirements specification, and, (2) its applicability in the industry, where the method has been found useful and not particularly onerous.

The content of this Appendix has been published in the *Journal of Computer Science and Technology* (JCST 2018) [RLRC18].

A.1 INTRODUCTION

Representing software requirements is a largely treated topic in literature, and a variety of methods, techniques and approaches have been proposed and applied in different domains. Among them, use cases are a widely used technique to specify the purpose of a software system, and to produce its description in terms of interactions between actors and the subject system [Coc00].

However, some common problems may emerge even while trying to explicit requirements by means of use cases. Ambiguities, incompleteness, and inconsistencies may in fact cause difficulty in requirements comprehension and, consequently, defects in the software system under development.

Screen mockups (also known as user interface sketches, user interface prototypes or wireframes) can be instead used for improving the comprehension of functional requirements, for prototyping the user interface of a subject system [HS91, O'D05], and for representing the non-functional requirements concerning the user interface [FNB07].

However, a drawback in enriching the use cases with the screen mockups is the burdensome task of guaranteeing the *consistency* between the graphical representation of the screen mockups and the textual descriptions of the use cases.

As a matter of fact, the consistency between screen mockups and use cases cannot be guaranteed if the former are not *disciplined* by a structure or by some rules. Requirements specifications based on use cases may in fact be scarcely structured, (e.g., composed of lists of freely formed natural language sentences), or they may be presented as quite detailed and structured templates (for example, Cockburn's [Coc00]), or even expressed through UML models [AR02] or formal specifications [CR04].

A good compromise can be represented by disciplined natural language specifications, where the text must follow very detailed and stringent patterns [RLRA12, LRRA12] and the screen mockups are used to clarify the steps of the scenarios.

For this reason, a method has been conceived for describing and refining requirements specifications based on *disciplined use cases* and screen mockups: DUSM (Disciplined Use cases with Screen Mockups).

The term *disciplined* means that a use case is: (1) characterized by a stringent template and complemented with a glossary to reduce ambiguities; (2) aligned with the screen mockups that will help in functionalities understanding; and, (3) able to help the requirements analyst to detect errors, incompleteness, bad smells (e.g., unused elements), and bad quality factors (e.g., too many scenario extensions, or, too many steps in a scenario) in the requirements specification, thanks to a list of well-formedness constraints.

Screen mockups are quite common in many IT (Information Technology) companies and several proposals are emerging to integrate/use them in conjunction with use cases (or, more in general, with requirements) [RGR+14, ZCC03].

DUSM fully and precisely integrates screen mockups with textual requirements. In the method, screen mockups are not only adornments for textual requirements specifications, but are artifacts (1) made consistent with the specifications by following a set of well-formedness constraints, and (2) checked against a list of possible bad smells that can lead to the detection of ambiguities, inconsistencies and incompleteness in the specifications. Another important aspect is that the design of screen mockups is based on screen mockup templates, which uniquely represent all the different and main aspects of a system GUI. Hence, there is a reduction in the effort needed to produce the screen mockups, since it depends on the number of the screen mockup templates instead on the higher number of the use cases steps.

This Appendix is organized as follows. Section A.2 describes the process adopted by DUSM for specifying and refining requirements. Section A.3 describe the requirements specification based on disciplined use cases and screen mockups. Finally, Section A.4 shows DUSM in practice, applied on a real case study.

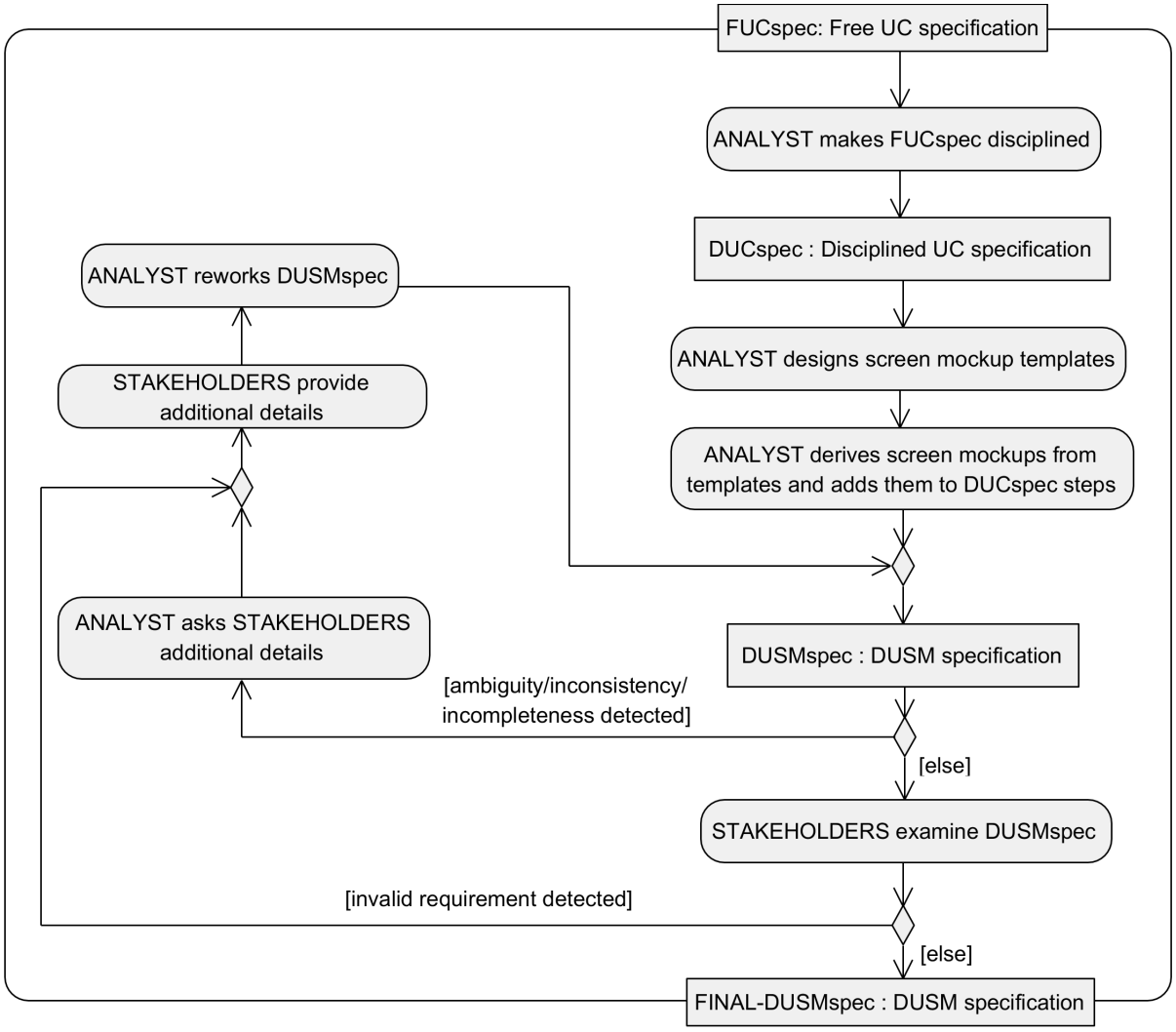
A.2 THE METHOD

The starting point of DUSM is what is called a *free use cases specification* (see FUCspec in Figure A.1), i.e., a use cases specification based on whatever template (for example, the one proposed by Cockburn [Coc00]), in general allowing a lot of freedom. In case not yet available, the free specification may be easily produced by stakeholders or domain experts with or without the assistance of the analyst.

Once the free specification has reached a stable form, the analyst may render the use cases disciplined (see DUCspec in Figure A.1), design the screen mockup templates, derive from them all the needed screen mockups, and add them to the use cases (see DUSMspec in Figure A.1). A screen mockup template is a way for enforcing a standard layout and look and feel across multiple screen mockups generated starting from it (jump to Section A.3.5.2 of this Appendix for more details).

Finally, the analyst will verify that all the well-formedness constraints advocated by DUSM are verified. The result of such activity is (often) the detection of problems in the specification, that can be classified in: *inconsistencies* (i.e., two different points of the specification express two contrasting statements about something), *ambiguities* (e.g., the specification uses words without stating their precise meaning relying on some common, but not always shared, understanding), and *incompleteness* (i.e., it is not possible to have a clear understanding of how the system should work because some parts are not properly specified). Notice that even the addition of the screen

Figure A.1: DUSM method as a UML activity diagram.



mockups may generate many questions about the system under specification and reveal undetected problems. In all these cases, the analyst should ask the stakeholders and/or the domain experts additional details to obtain a better specification, following a reworking phase of the previously produced one, where all the relevant changes are implemented, such as adding/removing/refining elements (e.g., some steps may need to be restructured, hence requiring further screen mockups).

Once the analyst has terminated this activity, the resulting disciplined specification enriched by screen mockups (i.e., DUSMspec in Figure A.1) may be given to the stakeholders to get the final approval. They will have no problem in reading and understanding the specification, since it is essentially structured natural language text. Moreover, the presence of the screen mockups provides a kind of paper prototyping, allowing them to validate also the user interface. Any

change request may be easily processed by the analyst, because the strong structuring of the DUSM specification offers a good support to propagate the changes on the whole specification. This characteristic combined with a better understanding of the entire specification will be valuable also in case of future evolution of the system requirements.

A.3 DISCIPLINED REQUIREMENTS SPECIFICATION

Figure A.2 shows the form of the DUSM requirements specifications by means of a meta-model presented by a UML class diagram. A *requirements specification* consists of a UML use case diagram, a description of each use case appearing in that diagram, and a glossary that lists and makes precise all the terms used in the use cases. The use case diagram has been included since it is really valuable for summarizing use cases, actors and their mutual relationships. Moreover, it is also quite simple to be explained and produced. It is important to highlight that the components of DUSM requirements specification are all well-known (e.g., use case diagram). What is novel here is how they are combined together (e.g., use cases and screen mockups) by means of rules and constraints.

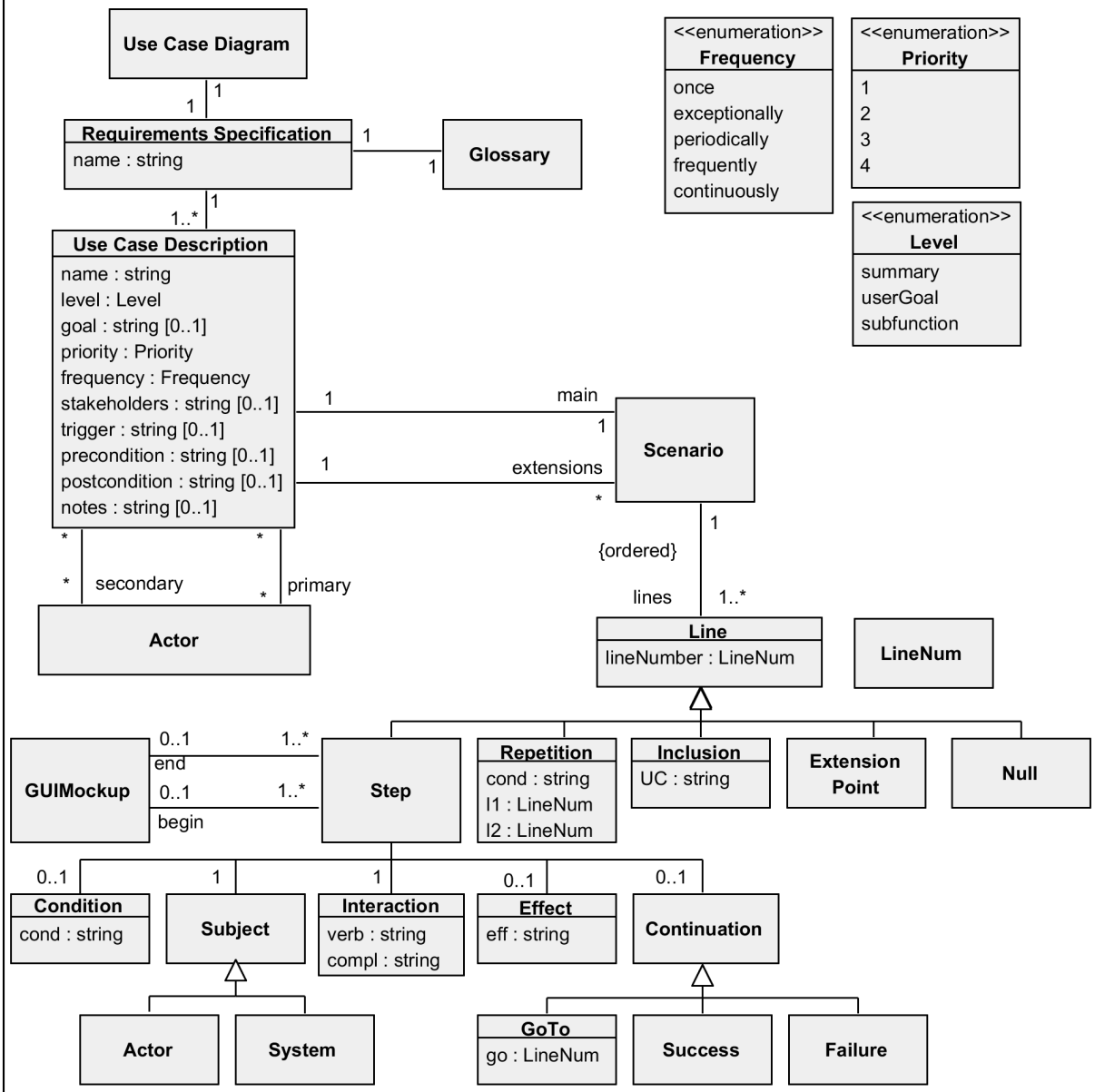
Even though some of the parts of the requirements specification shown in Figure A.2 are well-known, for the sake of completeness, in the following, all of them are briefly described, using the ACME case study (more detailed in A.4) as a running example. In this way, the novel aspects of DUSM are shown and explained.

ACME is a multi-users resource management system organized in functional areas, where users are assigned to tasks called jobs and can perform some actions, based on their access levels to the system functional areas, to complete them. An access level (**hidden** for a not accessible functional area, **view** for a read-only one, and **edit** for read-write permissions) is linked to a user through the unique group (s)he is assigned to. ACME works on two different levels of security: *enabled* and *disabled*. When security is *enabled*, each user must authenticate (with username and password) to perform any task. In case security is *disabled*, logging is not needed. A group in ACME can be thought of as a role which establishes all possible operations that can be completed. ACME presents two default groups: Administrator and External User. Each of these groups has different access levels for each of the ACME functional areas. An Administrator can CRUD (Create, Read, Update, Delete) any group, job and user and (s)he is responsible for the users assignments to groups. Instead, an External User can perform some actions depending on the access levels assigned to him/her; differently from the Administrator group, access levels for External User are not given by default. Every group and user is identified by a unique name. Among its attributes, a job is characterized by: a short description, a creator, and a timestamp of creation/update. An example of job may be the reservation of a room or the reporting of a bug, depending on which context ACME is applied. The user who

owns the rights (i.e., the access level for a functional area) could act like an Administrator for a specific job.

The term System is the following is used to denote a generic software system, while ACME is the more specific case study.

Figure A.2: DUSM requirements specification meta-model.



A.3.1 Use Case Diagram

The *use case diagram* summarizes the System *use cases*, making clear which *actors* take part in them. The actors are distinguished in: *primary*, those having goals on System, i.e., entities obtaining value from interacting with the System, and *secondary*, those over which the System has goals, i.e., entities supporting System in creating value for primary actors. Sticky-man icons are used for visually representing the primary actors, and boxes are used instead for the secondary actors.

Use cases are classified with respect to their granularity level: *summary* (representing a goal of the System), *userGoal* (representing functionalities from the user perspective), and *subfunction* (moving out an isolated part of a scenario to a separate use case). The granularity of a use case is depicted in the use cases icon (i.e., ellipses) of the UML use case diagram by means of three corresponding stereotypes. However, to improve readability, it can be omitted in case all the use cases have the *userGoal* granularity level.

Inclusion and *extension* relationships between use cases may appear in the use case diagram, as well as specialization relationships between actors, represented by the classic arrow with closed head. The inclusion relationship specifies that one use case includes the functionality of another use case mainly for reuse purposes, while the extension relationship specifies that one use case (extension) extends the behavior of another use case. Instead, the specialization relationships determines that, if actor A1 specializes actor A2, then A1 can take part also in all the use cases in which A2 takes part.

The list of well-formedness constraints and bad smells related to use case diagrams, proposed by DUSM, is shown in Figure A.3.

A fragment of the use case diagram part of the ACME requirements specification is shown in Figure A.4 (the complete version, amounting to 31 use cases, can be found in the ACME disciplined specification [RC15]).

In the case of ACME, there are three primary actors (Administrator, External User, and User), and no secondary actors; notice that both Administrator and External User specialize User, thus both may take part in all the use cases of User.

The level of all the use cases is *userGoal*, then for improving the readability of the diagram the corresponding stereotype is omitted.

A.3.2 Glossary

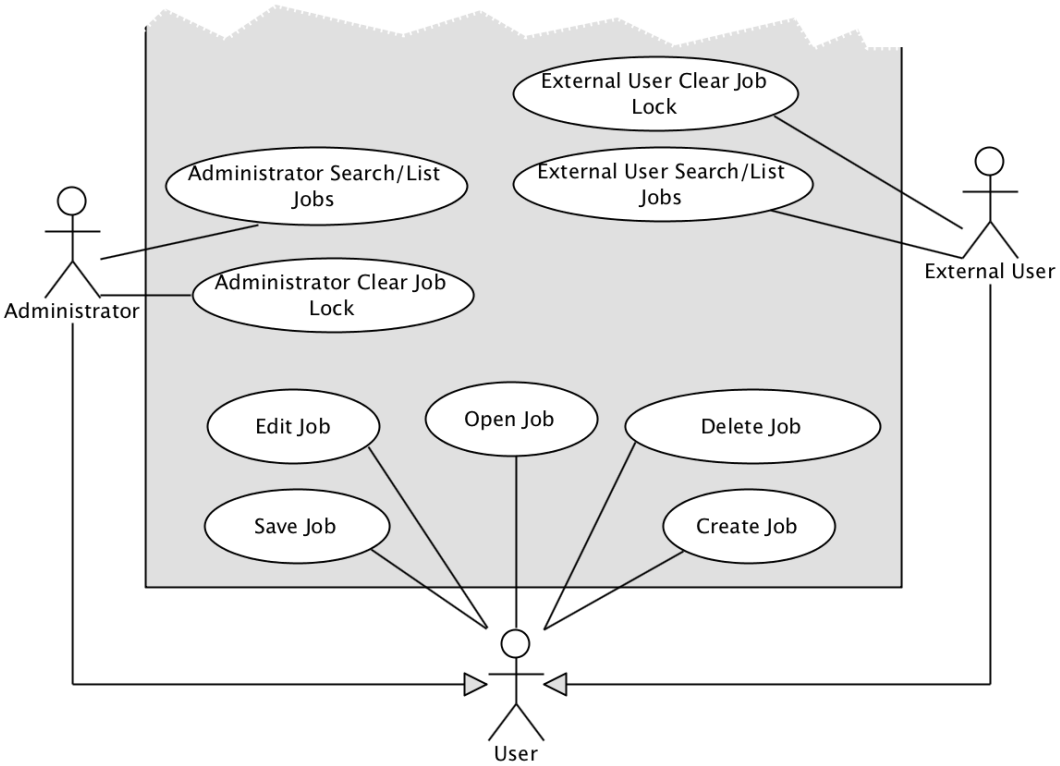
The *glossary* is a list of entries, each one consisting of the name of the defined term, and of a corresponding short description. The glossary entries are distinguished in those relative to *data entries* (e.g., credit card data, order info), and those about the attributes abstractly describing

Figure A.3: DUSM Use Case Diagram: Well-formedness constraints and bad smells.

Well-Formedness Constraints
- A use case cannot be included in a lower granularity level use case.
  WHY: *Violation of the meaning of the levels*
- A subfunction use case must be included at least in a userGoal use case.
  WHY: *Violation of the meaning of the subfunction level, since it cannot be a complete functionality*
- There must be at least a userGoal use case.
  WHY: *System must offer at least a functionality*
- The transitive closure of the inclusion relationship among use cases must be anti-reflexive.
  WHY: *There cannot be undefined use cases*
- The transitive closure of the extension relationship among use cases must be anti-reflexive.
  WHY: *There cannot be undefined use cases*
- The transitive closure of the specialization relationship among actors must be anti-reflexive.
  WHY: *There cannot be undefined actors*

Bad Smells
- A subfunction use case included only once in another higher level use case is suspicious.
  WHY: *This case is sensible if it has been introduced only for shortening the scenarios of another use case*
- A summary use case not including any other use case is suspicious.
  WHY: *If the use case is that simple, it should have the userGoal level*

Figure A.4: ACME requirements specification: Use case diagram fragment.



the state of System, indicated as *system attributes* (e.g., names of the registered clients, items currently in the catalog).

The main objectives of the glossary are to: *(i)* shorten use cases; *(ii)* reduce ambiguities (e.g., to avoid using different ways to refer to the same entity); and, *(iii)* clarify the meaning of the steps of a scenario (e.g., a richer description of the various entries could be given). The well-formedness constraints and the bad smells that must be considered for glossary are shown in Figure A.5.

Figure A.5: DUSM Glossary: Well-formedness constraints and bad smells.

Well-Formedness Constraints
- The name of an entry in the glossary must be unique.
  WHY: *It must be possible referring it in the use cases descriptions*
- Each entry in the glossary must appear at least once in a use case description.
  WHY: *Otherwise, it is useless and should be eliminated*

Bad Smells
- A glossary not containing system attributes is suspicious.
  WHY: *System not using any persistent data would provide only trivial functionalities, e.g., a converter from different measure units*
- A glossary not containing data entries is suspicious.
  WHY: *System using only basic data as numbers and strings would provide only trivial functionalities, e.g., a converter from decimal to roman numbers*

Figure A.6 shows a fragment of the glossary ACME requirements specification, where the entry names are written using a specific font (the complete one, amounting to 28 entries, can be found in ACME disciplined specification [RC15]). References to a glossary entry, in the form of entryName*, can be used in the definitions of other entries (e.g., see Job Info* in the definition of Jobs), in the use cases steps, or in the other parts of the use cases descriptions. References to glossary entries are case insensitive and can be replaced by a declension to adapt them to the sentences.

Figure A.6: ACME Requirements Specification: Glossary fragment.

Data Entries
- Access Level: defines the permissions to access the functional areas* of ACME. It can be: *hidden* for a non-accessible area, *view* for a read-only one, and *edit* in case of read-write permissions.
- Group Name: unique identifier built out of up to 32 characters (letters, numbers, separators such as dash or underscore, and blank spaces among them)
- Job Info: the data characterizing a job (short and long description, creator, title, issue, type, related module, creation/update timestamps)
- Search Criteria: the information given by a user to find some jobs (short description, creator, creation /modification date range, type, title, module, issue)

System Attributes
- Administrator Logged: a boolean value, true iff there is a logged administrator
- Jobs: all the existing jobs characterized by Job Info*

A.3.3 Use Cases Descriptions

A *use case description* consists of general information (e.g., name, level and goal), plus a set of scenarios (see Figure A.2). The *main success scenario* describes the basic execution of the use case, whereas the *extensions* (any number, also none) are scenarios defining all the other possible executions of the use case.

The information about a use case description are:

- ***Name***: a verbal phrase in the form of present infinite without the "to", that identifies the use case (e.g., Open Job). Sometimes it may be helpful to explicit the primary actor in the use case name, especially when several use cases share the same name but involve different primary actors and scenarios (see, for example, Figure A.4).
- ***Level***: i.e., summary, userGoal, or subfunction, as introduced in A.3.1.
- ***Goal***: describes in a detailed way the aim of the use case (optional).
- ***Priority***: expresses the impact that an incorrect or lacking implementation of the use case has on the System. Four values are sufficient, ranging from 1 (higher) to 4 (lower), where: priority equals to 1 means that System is no longer operative and no other ways to perform the supported activities exist; priority equals to 2 means that System is no longer operative, but the supported activities can be manually performed; priority equals to 3 means that System is operative, but one or more main functionalities are not available; priority equals to 4 means that System is operative, but one or more secondary functionalities are not available.
- ***Frequency***: expresses how much frequently the functionality described by the use case will be used. The possible values are: *once*, *exceptionally*, *periodically*, *frequently*, and *continuously*.
- ***Stakeholders***: defines who have the right or the possibility to say something about the functionality described by the use case (optional).
- ***Trigger***: defines which event starts the use case (optional).
- ***Primary and Secondary Actors***: introduced in A.3.1.
- ***Pre/post condition***: states what is about the current state of the System before the execution/ after the successful execution of the use case (optional). They should be expressed using the System attributes introduced in the glossary.
- ***Notes***: comments to the use case, usually to record why something is made in some way (optional).

The list of well-formedness constraints on the use case description is shown in Figure A.7.

Figure A.8 presents a simple use case description; it refers to the use case Administrator Search/List Jobs of the ACME system. Its level is userGoal, and its goal is to allow the primary actor

Figure A.7: DUSM Use Cases Descriptions: Well-formedness constraints.

Well-Formedness Constraints
- There should be a description for each use case appearing in the use case diagram and vice versa.
  WHY: *To guarantee the consistency between the use case diagram and the other parts of the specification*
- If a use case C is linked to actor A in the use case diagram, then A should appear in the actors part of the description of C (primary if its icon is the sticky-man, secondary otherwise), and vice versa.
  WHY: *To guarantee the consistency between the use case diagram and the use cases descriptions*
- Each term referenced in a use case description (i.e., term*) must appear in the glossary.
  WHY: *To guarantee that each term definition has been used and to prevent different wordings usages*
- Each actor of a use case must appear at least once in its scenarios, and vice versa.
  WHY: *To guarantee the consistency between the use case information and its scenarios*

Administrator to examine the existing jobs. The information part is quite standard and does not need a detailed comment, and this use case has just one extension. Underlined terms represent hyperlinks to screen mockups, while a term followed by * refers to an item of the glossary, partially shown in Figure A.6.

More complex use cases can be found in the ACME disciplined requirements specification [RC15].

Figure A.8: Administrator Search/List Jobs disciplined use case with linked screen mockups.

Use Case Administrator Search/List Jobs

Level: userGoal
Goal: The Administrator searches the jobs that are currently in the system.
Priority: 1
Frequency: frequently
Stakeholders: The company that intends to sell the software and the future users.
Primary Actor: Administrator
Preconditions: Administrator Logged* is true.
Main Success Scenario:
MainMockup
1. The Administrator requests to list the jobs.
2. ACME displays all Jobs* with their characterizing Job Info*.
JobsListMockup
3. The Administrator enters the desired Search Criteria*, then requests to search.
JobsListFilledMockup
4. If there is at least one job satisfying the Search Criteria*, then ACME lists all jobs matching it.
JobsFoundMockup
Extensions:
4a.1. If there is no job satisfying the Search Criteria*, then ACME informs the Administrator that no job is found. The use case fails.
NoJobsFoundMockup

A.3.4 Scenarios

The abstract structure of a *scenario* is presented in Figure A.2 (see Scenario class and associated classes). In particular:

Scenario: a sequence of numbered and ordered ***lines***, where each ***line*** is either a step, an indication of a repetition of some lines, an inclusion of another use case, an extension point or even a ***null*** line, which means a line corresponding to do nothing.

Line number: allows to uniquely identify each line of a scenario.

Step: describes an interaction between the System and one of the actors of the use case. It has the form: [If *condition*, then] *subject interaction*; [*effect*] [*continuation*].

where

- [. . .] means that . . . is optional.
- *condition* is a natural language fragment stating the condition under which the step may be executed. It is optional. If absent, it is intended as the always true condition. It should concern the System state attributes (i.e., the current state of System), and the data appearing in the interaction and effect part of the current step or of the previous ones; thus, it will be expressed using the terms introduced by the glossary.
- *subject* may be either an actor (primary or secondary) of the use case or the System. The System should be indicated with the same name used in the system box in the use case diagram.
- *interaction* is a sentence describing either what flows from the actor towards the System or vice versa. It must have the form "*verb complements*". The complements may be the System itself, an actor, System state attributes, and the data appearing in the preceding steps.
- *effect* is a sentence written in the passive form without explicit "by clause" (e.g., omit "by ACME" in the effect "a new user characterized by User Info* is added to Users* by ACME") describing a transformation of the System state attributes using the data appearing in the interaction; thus, again, it will be written using the terminology introduced in the glossary. It is optional. If absent, then the step does not influence the System attributes.
- *continuation* defines how the use case flow continues after the end of the step. It may have one of the following forms:
  - "The use case continues to *lnum*", where *lnum* is the number of a line of the use case; this case is named *GoTo* continuation in Figure A.2,
  - "The use case fails." and "The use case ends with success." for marking the end of the use case, distinguishing whether it is a success or a failure. For the sake of readability, "The use case ends with success." is usually omitted.

The continuation is optional. If absent, then the use case continues to the next line.

Repetition: a natural language fragment having form: "The lines from $lnum_1$ to $lnum_2$ are repeated until *cond*", where $lnum_1$ and $lnum_2$ are the numbers of two lines of the scenario including the repetition, and *cond* is a condition like the one that is part of a step. Repetitions allow to describe scenarios of unbound length.

Extension point: denotes where the behavior of an extending use case (i.e., a use case related by the extension relationship in the use case diagram) will be inserted.

Inclusion: written by reporting the name of the included use case, that should be linked to the described use case by the inclusion relationship in the use case diagram.

Null: a line where nothing is done (i.e., neither an actor nor the System does something); it is needed for representing particular flows of activities. For example, in ACME a user could optionally choose to view some details of a job while performing some other activities; since both the alternatives of viewing job details and not viewing them (i.e., do nothing) may affect ACME response to the user in different ways, two scenarios have been modelled.

Extensions: of a use case, see Figure A.2, are scenarios defined modifying an existing one, by giving a different sequence of lines starting from a given line (the extended line), where the lines of a use case are identified by *line numbers*. The lines of the main success scenario are labelled by natural numbers (i.e., *1*, *2*, *3*, ...). The lines of the first scenario extending a line whose number is *X* are numbered with *Xa.1*, *Xa.2*, ..., those of the second scenario *Xb.1*, *Xb.2*, ..., and so on.

For what concerns scenarios well-formedness constraints and bad smells, refer to Figure A.9.

A.3.5 Screen Mockups

Screen mockups are drawings that show how the user interface of a System is supposed to look during the interaction between the System and the human actors. They may be very simple, just to help the presentation of the user-system interactions, or more detailed, with rich graphics, whenever specific constraints on the graphical user interface need to be expressed (e.g., requiring to use specific logos or brand related colors) [O'D05].

Mockups can be used in conjunction with use cases, associating them with the steps of the scenarios, to improve the comprehension of functional requirements, and to achieve a shared understanding on them. At the same time, screen mockups allow to express and improve the comprehension of the non-functional requirements concerning the user interface [FNB07], thanks to the freedom in choosing the most effective way to represent them.

DUSM suggests to associate one or two screen mockups with each step of a scenario (see Figure A.2), where a human actor is involved.

Figure A.9: DUSM Use Cases Scenarios: Well-formedness constraints and bad smells.

Well-Formedness Constraints

- The subject of a step of a scenario different from System must appear among the use case actors.
  WHY: *To guarantee the consistency between the information part of a use case and its scenarios*
- Each System attribute listed in the glossary must both:
  – be updated in the effect part of at least a step of a use case scenario,
  – be read either in the condition or in the interaction or in the effect of at least a step of a use case scenario.
  WHY: *To guarantee the minimality of the specification. If it is neither read nor updated it is useless, and thus it should be removed. If it is updated and never read, it is either useless or the steps/use cases reading it are lacking. If it is read but never updated the steps/use cases modifying it are lacking*
- If a step has a condition *cond* different from true, then there should be some extensions starting from the same step with conditions $cond_1, \ldots, cond_n$ s.t. the logical disjunction of *cond*, $cond_1, \ldots, cond_n$ is true. It is possible to use the Null line in the case nothing is done for a certain condition.
  WHY: *To specify what* System *should do in all possible cases*
- If a use case C includes C1 in the use case diagram, then at least a line corresponding to "include C1" must appear in the scenarios of C, and vice versa
  WHY: *To guarantee the consistency between the use case diagram and the use cases descriptions*
- If a use case C extends C1 in the use case diagram, then at least a line corresponding to an extension point for C must appear in the scenarios of C1, and vice versa
  WHY: *To guarantee the consistency between the use case diagram and the use cases descriptions*
- The line number appearing in a GoTo continuation of a step (see Figure A.2) should refer to an existing line in the use cases scenarios.
  WHY: *There cannot be dangling GoTo references*
- The line numbers appearing in a repetition line should refer to existing lines in the same scenario.
  WHY: *There cannot be dangling repetition references*
- The success and failure continuation may appear only at the end of complete scenarios.
  WHY: *To express the final outcome of the various use cases executions*
- The failure continuation cannot appear in the main success scenario.
  WHY: *The main success scenario must describe a successful way to execute the use case*

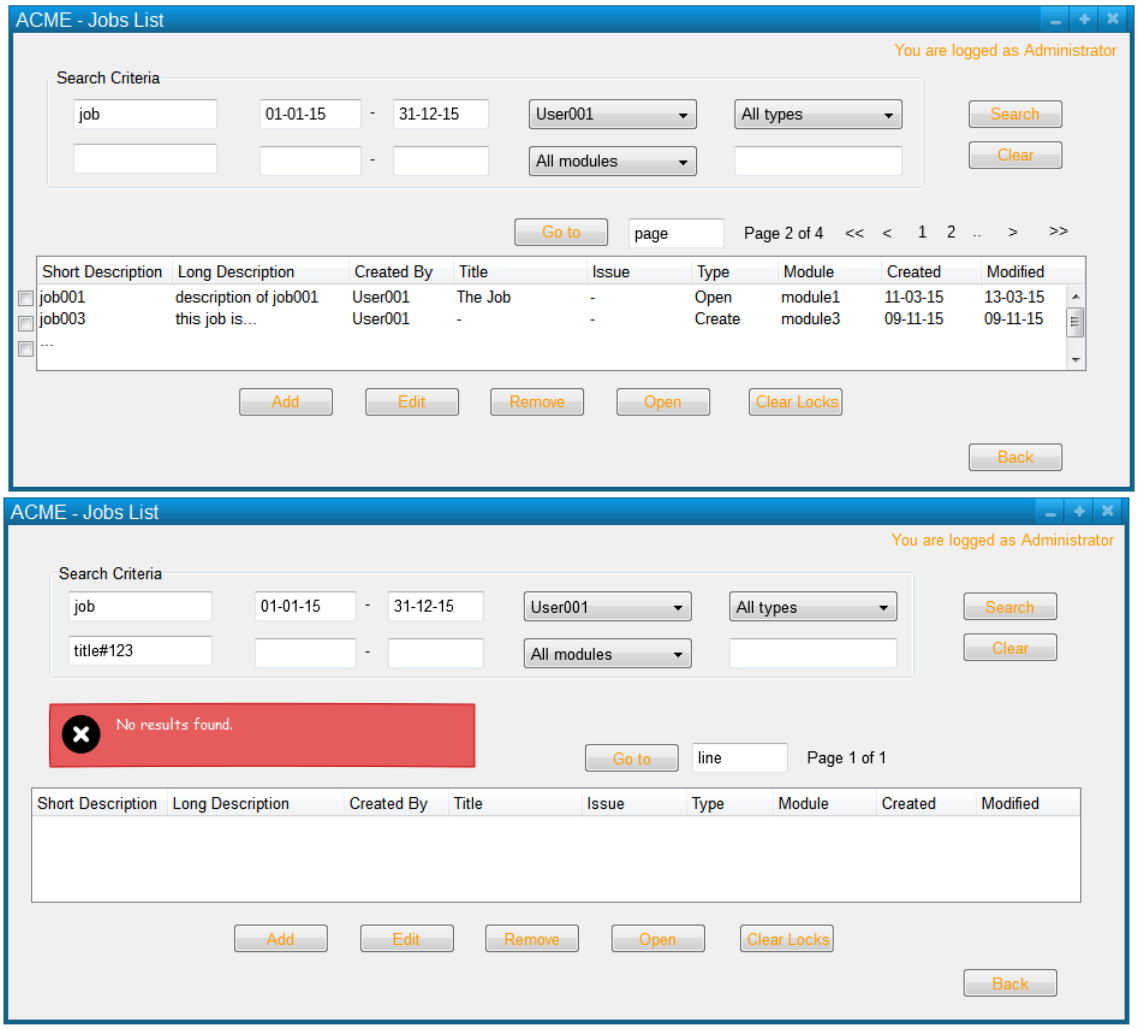
Bad smells

- A complete scenario without at least a step where the subject is System is suspicious
  WHY: *The System must provide some feedback to a request of an actor, unless the actor is not human*
- The case where the initial steps of a set of extensions to the same line do not have the same subject is suspicious.
  WHY: *Usually such cases correspond to awkward behaviors of the specified System.*
- An extension S_1 of a step *S* s.t. the subject, the interaction and the effect of S_1 coincide with those of *S* is suspicious
  WHY: S *and* S_1 *must be joined in a unique step by combining their conditions*

Figure A.10 shows two simple screen mockups associated with some steps of the Administrator Search/List Jobs use case (presented in Figure A.8) of the ACME application.

More precisely, let *S* be a step of a scenario:

Figure A.10: ACME Screen Mockups: JobsFoundMockup and NoJobsFoundMockup.



- If the subject of *S* is System, then at most one mockup (*end mockup*) may be associated with the end of *S*; it will show what can be seen on the System's GUI after the step execution
- If the subject of *S* is a human actor, then at most two mockups may be associated with *S*:
  - one at the beginning of *S* (*begin mockup*), which will show what the actor sees just immediately before to execute the step.
  - one at the end of *S* (*end mockup*), which will show what the actor sees just immediately before to complete *S* (e.g., before pressing the "send" button after having filled various fields/ticked some check boxes/opened some menus).

The begin mockups of the steps corresponding to extensions starting from the same step, obviously, must coincide.

A single mockup may be associated with various steps; examples are the begin mockup of different extensions, and the end mockup of a step coinciding with the begin mockup of the next step. See, for instance, JobsListMockup in Figure A.8 that is shared between steps 2 (as end mockup of a system's action), and 3 (as begin mockup of an actor's action). A single mockup can also appear in different use cases, as in the case of the one corresponding to the main window of a system.

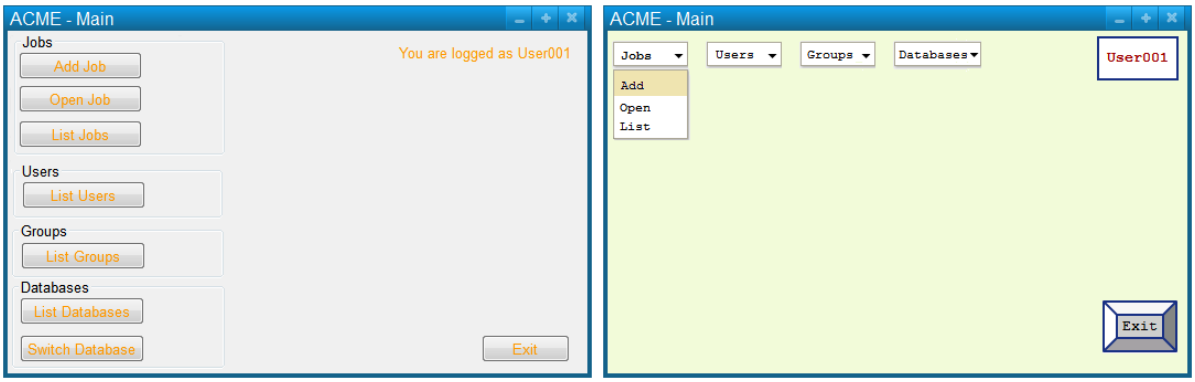
Any screen mockup associated with a step must be consistent with it, i.e., it should present the same informative content, otherwise the introduction of the mockups will be the cause of further ambiguities in the requirements specifications, instead of improving their quality. An example of inconsistency is the linkage of a screen mockup containing exactly two text boxes to a step having form "Insert name, birth date and sex". Similarly, the mockup associated with "System informs that a username must contain capital letters and digits only" cannot present a popup showing just a generic "Error" message.

DUSM provides detailed constraints (see Figure A.12) that guarantee the consistency between the steps and the associated mockups. These constraints may be also considered as guidelines to help the production of the mockups. However, the given constraints do not lead to a standard straightforward way to design the screen mockups to add to the steps. Clearly, if three entities are mentioned in a step, e.g., name, birth day and sex, the associated mockup should contains three widgets, which may be textboxes, drop-down menus, checkboxes, and so on. Moreover, the layout and the aspect of the mockup is entirely left to the analyst. This freedom in the design of the mockup(s) associated with a step allows to express the non-functional requirements on the application GUI.

For example, referring to the ACME case study, Figure A.11 on the left shows the main window, i.e., the begin mockup of many steps starting the main functionalities of the application, as originally designed. It is quite simple and offers buttons for calling all such functionalities. If, instead, the interface should adhere to an existing theme, the mockup could be drawn as the one on the right of Figure A.11, where functionalities are accessible through menus, using a different color theme and a different font.

Placeholders for the begin/end mockups will be inserted respectively before or after the steps in the use cases scenarios. Obviously, whenever the begin mockup of a step coincides with the end mockup of the previous step, its placeholder will appears only once in the scenario. Placeholders may be realized in different ways depending on the technology used to write the use cases (e.g., a link to a picture in a Word document, or a hyperlink in a HTML document). In Figure A.8, for example JobsFoundMockup and NoJobsFoundMockup are links to the pictures reported in Figure A.10. The use of the placeholders allows the readers of the use case to choose whether to examine the screen mockups or ignoring them, whenever interested only in the flow of the various steps. Instead, by replacing all the placeholders with the corresponding pictures, an

Figure A.11: Two different screen mockups for ACME main window.



alternative visualization of the use cases can be provided (see some examples in ACME disciplined specification [RC15]), corresponding to the so called “paper prototype” of a System.

Figure A.12: DUSM Screen Mockup: Well-formedness constraints and bad smells.

Well-Formedness Constraints

Steps with actor subject – Begin mockup

Let $S_1, \ldots, S_n$ *($n \geq 1$) be all the steps in the specification having an actor as subject s.t. their begin mockup is* M *(*$S_1, \ldots, S_n$ *may appear in different scenarios of even different use cases).*

- If the interaction part of some S_i ($1 \leq i \leq n$) refers to some communication from the actor to System, then *M* should show how it has been realized.
  e.g.,: n=2, S_1 ="User confirms", S_2 ="User refuses". Thus, "Confirm" and "Refuse" buttons appear in *M*;
- If *M* contains some means for realizing some communication from the actor to the System, then there should exists $1 \leq i \leq n$ s.t. S_i refers to such communication.
  e.g.,: *M* contains a "Confirm" button. Thus, the interaction part of S_i, for some i, should speak of confirmation;

Steps with actor subject – End mockup

Let $S_1, \ldots, S_k$ *($k \geq 1$) be all the steps in the specification having an actor as subject s.t. their end mockup is* M *(*$S_1, \ldots, S_k$ *must be steps having the same interaction part appearing in different scenarios of even different use cases).*

- If the interaction part of S_1 (that is coincident with those of $S_2, \ldots, S_k$) refers to some communication from the actor to System, then *M* should show how it has been realized.
  e.g.,: S = "User selects the amount to deposit". Thus, a text box filled with a given amount is shown in *M*;
- If the interaction part of S_1 (that is coincident with those of $S_2, \ldots, S_k$) includes a reference to some specific information (flowing from the actor to System), then such information must appear in *M*.
  e.g.,: S = "User selects category and number of nights". Thus, both the category and number appear in *M*.
- If *M* shows how some communication is going to be realized (from the actor to System), then the interaction part of step S_1 (that is coincident with those of $S_2, \ldots, S_k$) should refer to it.
  e.g.,: *M* contains a filled "password" field. Thus, the interaction part of S_i, for some i, should describe the insertion of password.

Steps with System subject – (End) mockup

Let $S_1, \ldots, S_m$ *($m \geq 1$) be some steps in the specification having System as subject s.t. their end mockup is* M *(*$S_1, \ldots, S_m$ *should be steps having the same interaction part appearing in different scenarios of even different use cases).*

- If some information appears in *M*, then it should be derived by the interaction parts of the previous steps or by System attributes.
  e.g.,: "You are logged as John Doe" message is shown in *M*. Thus, the name of "John Doe" is recoverable by System attributes or it is provided by the user in some previous step.
- If the interaction part of S_1 (that it is coincident with those of $S_2, \ldots, S_m$) refers to some communication from System to actor, then *M* should show how it has been realized.
  e.g.,: S = "System lists all the logged users". Thus, "The current logged users are:" string appears in *M* together with a list of users;

WHY: *To guarantee the consistency between the use cases descriptions and the screens mockups*

Bad Smells

- Having several screen mockups in the extensions and none in the main success scenario is suspicious.
  WHY: *It does not present user interface requirements in a systematic way; it is acceptable only in the case the mockups for the main scenario are obviously similar to those of another use case*
- Having few mockups distributed in almost every use case with human actors is suspicious.
  WHY: *In this way it is difficult to grasp a coherent set of requirements on the GUI. It is better to concentrate the effort and thus adding all the mockups to a small number of use cases*

A.3.5.1 How to Produce Screen Mockups

Although a number of tools for drawing screen mockups exist (see Table A.1 for a partial list), several professionals prefer to sketch screen mockups on paper. This kind of approach has some drawbacks, which are mainly related to the continuous evolution of requirements [LM95]. Mockups created with computer drawing tools mitigate this last concern, so making them a viable alternative.

Screen mockups could be also built using HTML, a programming language (e.g., Java), or an IDE (e.g., NetBeans, Visual Studio). These last choices have the benefit to reuse the source code of screen mockups later in the development phase and to obtain quite realistic mockups, while the main drawbacks concern the effort and the skills needed to build and maintain them: in fact, any kind of stakeholder (even without development experience) should be able to build mockups spending a few minutes [RST+10, SRT+13].

Therefore, the use of specific tools for drawing screen mockups represents a viable trade-off between sketching screen mockups on paper and implementing them. For instance, the screen mockups shown in Figures A.10 and A.11, and all the mockups appearing in the disciplined requirements specifications of the ACME [RC15] case study have been created using Pencil tool.

DUSM does not force to add all the possible mockups to all steps of the scenarios of all the use cases (see the multiplicities on the associations linking the mockups to the steps in Figure A.2); it is left to the analyst the decision to omit those not conveying any useful information (e.g., mockups associated with steps where the user interaction is trivial or too similar to already inserted ones). Furthermore, the relevance of the various use cases (expressed by the priority and frequency attributes) helps to select which ones should be enriched with screen mockups; for example, it is of scarce utility to produce the mockups for a use case whose frequency is "Once", whereas it is almost mandatory in case of "Continuously". Similarly, it is obviously better to add mockups to use cases with priority 1 than to those with priority 4.

Table A.1: A partial list of mockups drawing tools.

Tool	**URL**
Balsamiq Mockups	https://balsamiq.com
Pencil Project	https://pencil.evolus.vn
GUI Design Studio	https://www.carettasoftware.com
Axure	https://www.axure.com
Moqups	https://moqups.com
MockFlow	https://www.mockflow.com
Mockingbird	https://gomockingbird.com/home
BlueGriffon	http://www.bluegriffon.org

A.3.5.2 Relations Among Screen Mockups

The number of mockups associated with use cases scenarios are not a bunch of totally unrelated items. Quite naturally, they are organized in a forest-like structure where there is an arc from M_2 to M_1 if and only if M_2 is derived from M_1, i.e., M_2 has been produced by modifying M_1 (e.g., when a menu or an auxiliary window is opened, a text-box is filled, or a button become dimmed).

For example, in the use case Administrator Search/List Jobs shown in Figure A.8, the screen mockup named JobsListFilledMockup is derived from JobsListMockup by filling some text fields. Technically, there is a *dependency relationship* between the mockups, assuming the classical definition: M_2 depends on M_1 if and only if a modification in the latter leads to a modification in the former (e.g., if the layout is modified in M_1, then also M_2 should be modified).

Producing each tree in the mockups forest requires to express some (non-functional) requirements on the GUI, e.g., by deciding the layout and the color theme of a window. Thus, before to produce the mockups, all the main requirements on the GUI should be expressed by providing a *template* for each type of GUI that will be used, to later be able to produce the trees in the forest. Therefore, a template is just a screen mockup that uniquely represents a portion of the system GUI; it can be seen as a node of a tree where the children of that node are all the variations of their parent. All the templates should be obviously coordinated: for example, it is better to avoid to notify an error in a mockup using a popup and a message bar at the bottom of a window in another one.

So, to proceed to build the mockups, DUSM suggests to:

- determine the needed screen mockup templates and organize them with respect to the dependency relationship;
- produce the needed mockups modifying either a template or an already produced mockup.

The suggested way to produce and organize the mockups will be helpful in the case of evolution of the requirements specification (e.g., dependency will help to propagate any change in the requirements on the user interaction). Furthermore, the effort needed to produce the mockups is not depending linearly on the number of the use cases and on the number of their steps, but on the number of the templates, which is generally lower, i.e., on the variability of the System's GUIs.

A.4 ACME CASE STUDY

DUSM has been proposed and fine-tuned during several editions of a software engineering course at the University of Genova [ACRR07]. Each year, students had to realize a Java desktop application, whose requirements were given as a use case based specification. First, students had to model a design by means of UML, and then, they had to implement it in Java [ACRR07]. Initially, no screen mockups were used, and even if standard requirements on the GUI were

provided (i.e., usability requirements), the use cases often resulted difficult to understand and ambiguous. After the introduction of screen mockups, the number of misunderstanding about the use cases decreased. Then, DUSM has been applied also in real industrial contexts, to specify requirements from scratch and to refine already existing use case specifications.

In this section, DUSM is applied to an already existing free requirements specification of an application named ACME (the *object* of the study). The goal is restructuring the ACME specification [Ale04] (from free to disciplined) and, at the same time, answering the following two research questions:

RQ1: is the application of DUSM able to reveal and remove inconsistencies, ambiguities, and incompleteness from the original ACME free specification?

RQ2: is the application of DUSM able to produce a specification that is simpler w.r.t. the original ACME free specification?

The metric used to answer **RQ1** is the *number of inconsistencies/ambiguities/incompleteness revealed*, while the one used to address **RQ2** is the *percentage of verbosity reduction*. The verbosity reduction is evaluated in terms of words saved in the disciplined version w.r.t. the free version, clearly preserving the semantic content.

RQ1 concerns *quality of the entire specification* and *comprehensibility* factors. Indeed, removing inconsistencies/ambiguities/incompleteness from the specification should improve the comprehension as well as the quality of the entire document. Instead, the *simplicity* mentioned in **RQ2** mainly correlates with *comprehensibility*: a short specification is generally much more comprehensible than a long/verbose one. For the computation of this last measure, the introduced novel screen mockups and their content has not been considered.

A.4.1 ACME Free Specification

The ACME specification is suitable to be used as the object of the study for the following reasons:

- It has been produced by a professional developer (therefore, it does not originate from the academia);
- It is not trivial (the documentation is around seventeen-thousand words and 22 use cases [Ale04]);
- It has been produced following a quite formal template and revised many times (last version is 6.1, 2004), also after discussing with the client. So, this free specification should be already of a good quality, and thus a good test for seeing if DUSM helps to improve it; otherwise, the validation may be biased (e.g., starting from a very informal and never revised specification);

- The domain of ACME is easy to grasp, so there is no need of know-how of specialized domains to apply the method;
- The use case steps are complemented with the textual description of the GUI, thus ACME is an application where expressing the requirements on the user interactions is relevant.

The use cases in the ACME requirements specification are quite complex, with verbose and tricky scenarios built up by many alternative paths, long steps and references among them (i.e., *GoTo* continuation jumps). In many cases, the steps of the scenarios are made heavier due to the inclusion of detailed textual descriptions of the associated GUI, and of the data shared among actors and ACME.

Since the approach followed by the author of ACME specification does not consider the specialization between actors, when a use case has two primary actors, it may be either that they cooperate to realize the functionality described in the use case or that they both separately (thus, not simultaneously) take part in the use case. This feature clearly has worsened use cases understandability. Moreover, no use case diagram has been adopted to represent the use cases and their relationships, while only an incomplete and inconsistent use cases list has been provided.

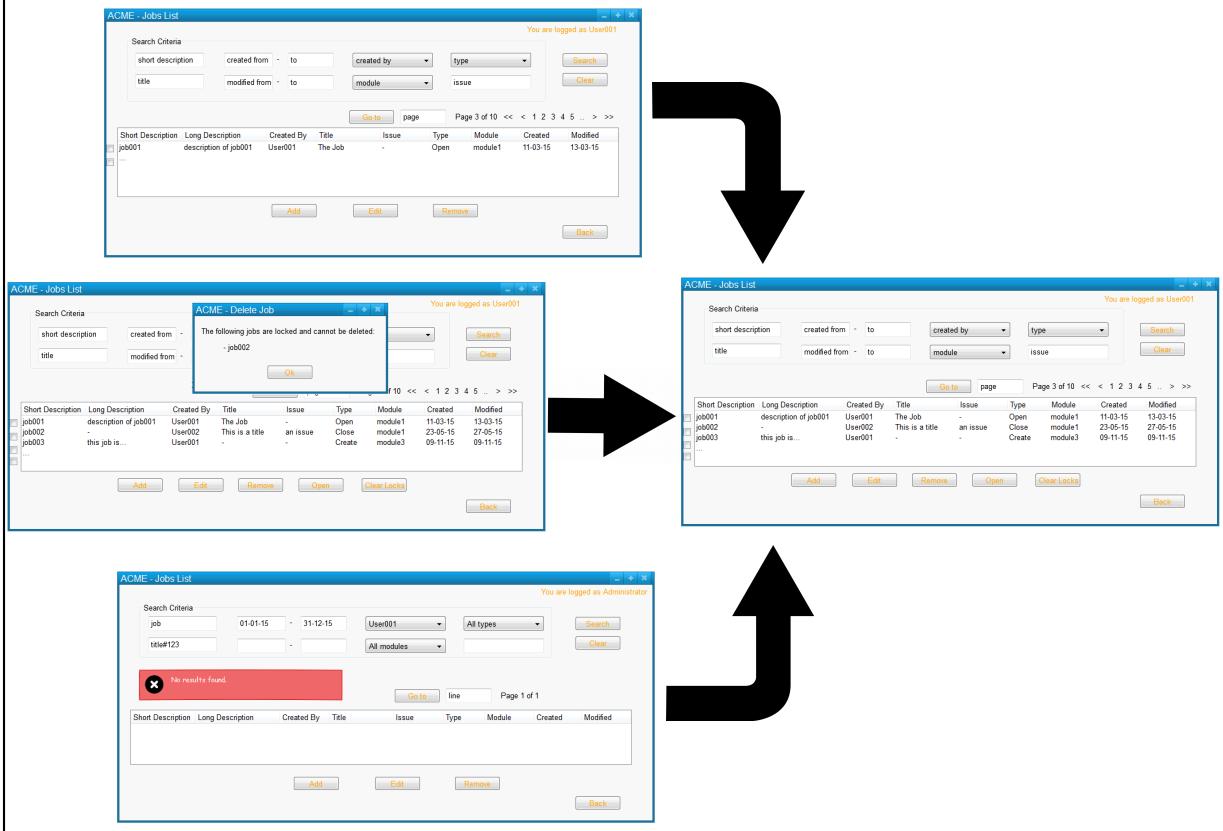
A.4.2 ACME Disciplined Specification

In accordance with DUSM activities of Figure A.1, ACME specification was restructured/refined by performing the following tasks: (1) glossary definition; (2) use cases description refactoring; and, (3) screen mockups integration. Moreover, during this restructuring task, the use case diagram was also inferred, as partially shown in Figure A.4.

The entries in the glossary were recovered from the existing use cases descriptions, focusing on *Notes*, *Pre conditions*, *Post conditions*, and scenarios steps. Each term was classified as *Data Entry* or *System Attribute*; a definition was formulated by looking in many different points of the original document, then references to the glossary entries were used in any place such entries were mentioned. See Figure A.8 for an example of glossary usage.

For what concerns use cases descriptions, their steps and scenarios were completely restructured because of their complexity. In some circumstances, new use cases emerged due to the separation of actors that were wrongly linked to a single use case. For example, Search/List ACME Jobs use case from original ACME specification shows both Administrator and External User as actors, so it was split into Administrator Search/List Jobs (see Figure A.8) and External User Search/List Jobs (see ACME disciplined specification [RC15] for further examples). Moreover, the steps were shortened by using the proposed form (i.e., [if *condition*, then] *subject interaction* ; [*effect*] [*continuation*].), along with the usage of the terms from the glossary. To further reduce the granularity of a step, a split into more steps was applied when needed (e.g., when a step includes more than a unique interaction).

Figure A.13: Screen Mockups Dependency: A ACME template (right) and some children (left).



At last, screen mockups were added in order to simplify those steps expressing a non-trivial user interaction (e.g., a step where a user has to fill a form), reducing verbosity and expressing a prototypical GUI. For drawing the screen mockups, the Pencil tool was used (see Table A.1 for a partial list of possible tools). At start, the basic templates were identified and drawn (amounting to 15), from which some variations that differed in some details were produced, once needed, by extending, changing or removing the visual parts of the original ones. Figure A.13 shows an example of the dependency relation that exists between a template and some possible variations. In this example, the template (right) was the first mockup encountered during the restructuring process. The mockups successively found as similar to the first one (e.g., same functionality of ACME, but different permissions/state) were associated with it in a dependency relationship. It is interesting to notice that, in some cases, this relationship may involve a “downgrade” of the original template (i.e., something is removed), an “upgrade” (i.e., something is added) or both.

Notes were taken any time problems were encountered in the original ACME requirements specification (i.e., ambiguities, inconsistencies, and incompleteness). Moreover, when a question for the stakeholders had to be answered to solve the problem, the role of the stakeholders was played to produce a possible answer.

Figure A.14: Comparison between free (left) [Ale04] and corresponding disciplined (right) [RC15] fragment of Search/List ACME Jobs use case.

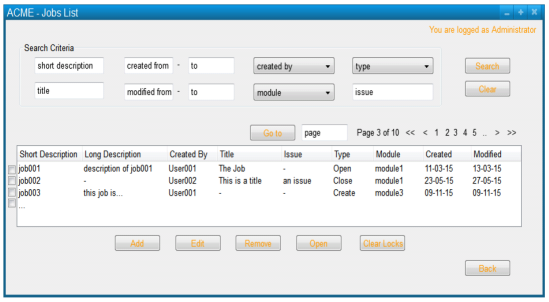
…

1. The user selects the "List Jobs" option
2. The system displays a window that lets the user filter job listings by:
    1. Short description
    2. Title
    3. Created (date range)
    4. Modified (date range)
    5. Created By
    6. Module
    7. Type
    8. Issue
3. The user enters the desired search criteria, then requests to search
4. The system lists the found jobs, sorted by the following fields:
    1. Short description
    2. Long description
    3. Created by
        1. If the user account has been deleted from the system, display Unknown User instead
    4. Title
    5. Issue
    6. Type
    7. Module
    8. Created (date)
    9. Modified (date)
5. The user can sort the data by any of the displayed columns
6. The user can use standard "VCR buttons" to move to the First, Previous, Next, Last and User-specified page number

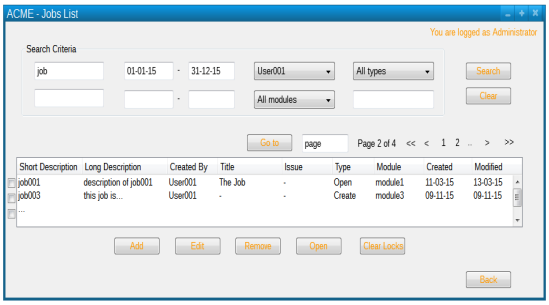
…

…

1. The Administrator requests to list the jobs.
2. ACME displays all Jobs* with their characterizing Job Info*.



3. The Administrator enters the desired Search Criteria*, then requests to search.
4. If there is at least one job satisfying the Search Criteria*, then ACME lists all jobs matching it.



…

An example of the application of DUSM to ACME is shown in Figure A.14. It compares a fragment of the Search/List Jobs use case of the ACME *free* specification [Ale04] (left) against the corresponding disciplined Administrator Search/List Jobs use case of the ACME *disciplined* specification (right). To make the free use case disciplined, a split of the two involved actors (Administrator and External User) was needed.

The complete ACME disciplined specification can be viewed in [RC15].

A.4.3 Results

For answering **RQ1**, during the application of DUSM to ACME case study, a set of assumptions were made and a number of inconsistencies, ambiguities and incompleteness were detected, classified and noted down.

Figure A.15: Inconsistencies/ambiguities/incompleteness found in ACME free specification (**RQ1**).

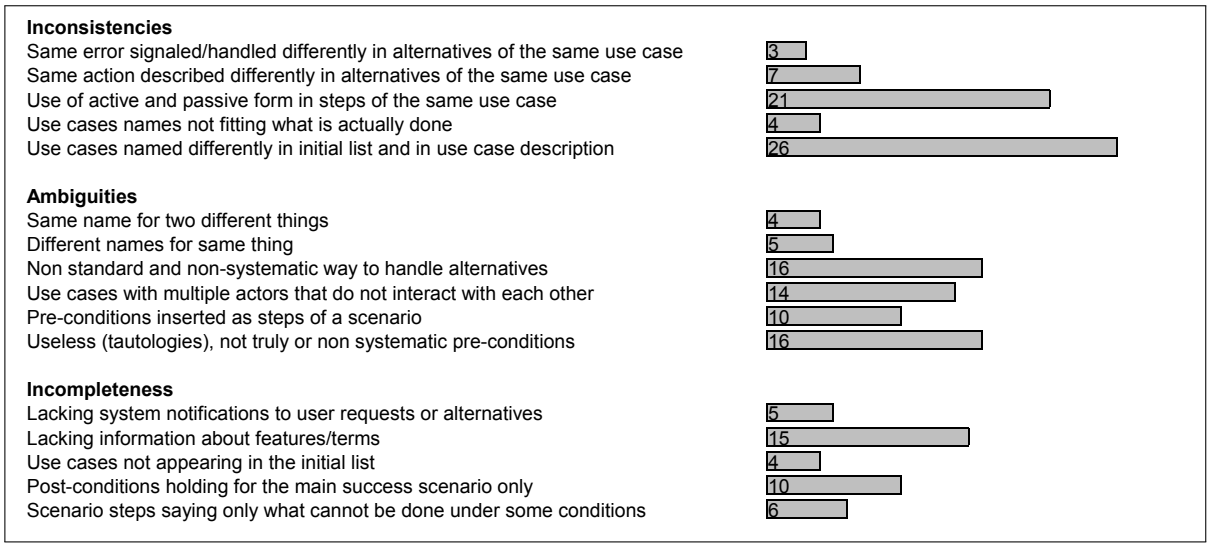


Figure A.15 lists all the various kinds of inconsistencies, ambiguities and incompleteness found, along with the number of their occurrences. This list has been generated thoroughly inspecting the original ACME specification.

For example, there are 3 occurrences of *Same error signaled/handled differently in alternatives of the same use case* in the **Inconsistencies** category, which means that the original specification signaled or handled errors in different ways among the various alternatives of a given use case (e.g., sometimes an error message was shown and sometimes was not; moreover, the content of that message was not always the same). As another example, in the **Ambiguities** category, *Same name for two different things* was detected 4 times during the analysis (e.g., both External User and User terms were used to indicate the same actor).

One of the biggest problems faced to make ACME disciplined was the lack of information to clarify the terminology; in fact, *Lacking information about features/terms* in the **Incompleteness** category was detected 15 times. Incompleteness, in particular, left with open questions that had to be answered in order to make the ACME specification disciplined, since the stakeholders cited in Figure A.1 were not available. For this reason, a list of open questions (turned, then, into **Found problems**) and provided answers (turned into **Possible solutions**) was generated, as shown in Table A.2.

For instance, *Unclear core functionalities* entry means that some main tasks, in particular the job-related ones, were left unspecified. A possible solution was proposed trying to understand and define the term “job”, in accordance with the information gathered through the documentation.

Table A.2: Found problems and possible solutions to discipline ACME free specification.

Number	Found problem	Possible solution
1	Unclear actors in use cases	If it is unclear who is acting or what are the differences in use cases scenarios, all actors interact in the same way.
2	Unclear permissions in use cases interactions	If no restriction is made explicit in use cases scenarios, all actors can complete any kind of interaction
3	Unclear core functionalities	If a core functionality is unclear, it is modeled in accordance with the information gathered in the documentation
4	Terminology abuse	If more terms share the same semantics, just one of them is used in use cases scenarios
5	Terminology incompleteness	If terms are referenced in use cases scenarios but never described, they are removed or simplified
6	Lacking of alternatives in use cases scenarios	If alternatives are lacking in use cases scenarios, further steps are added to complete all possible alternatives

Similarly, the *Unclear actors in use cases* entry means that there were some use cases not clarifying which actor could interact with ACME and which functionalities were enabled for them; in this case, all the actors were left to complete any kind of interaction with no restrictions.

For answering **RQ2**, the original ACME specification was compared with the final one. Summarizing, the number of disciplined use cases was increased to 31 (+9), since some original use cases were split between the involved actors (see ACME disciplined specification [RC15]), whereas the total amount of entries in the glossary amounted to 28. A total of 114 screen mockups were produced, starting from a baseline of 15 templates. Considering only the textual part of the specification, the DUSM adoption downsized the original ACME free specification by about 63%, from approximately seventeen-thousand (exactly, 16917) to less than seven-thousand words (exactly, 6221).